Universidad Zaragoza

Trabajo Fin de Grado

Estudio de Entornos de Ejecución Confiables sobre la Arquitectura RISC-V

An Study on Trusted Execution Environments in the RISC-V Architecture

Autor

Luis Felipe Nonay Serrano

Directores

Rubén Gran Tejero

Darío Suárez Gracia

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2023

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Luis Felipe Nonay Serrano,

con nº de DNI 17460373S en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado en Ingeniería Informática, (Título del Trabajo) Estudio de Entornos de Ejecución Confiables sobre la Arquitectura RISC-V

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 31 de Agosto de 2023.

Fdo: Luis Felipe Nonay Serrano

AGRADECIMIENTOS

En primer lugar, quiero expresar mi más profundo agradecimiento a Rubén Gran y Darío Suárez, gracias por vuestra dedicación, ha sido un verdadero placer poder aprender con y de vosotros durante el grado, y finalmente poder culminarlo trabajando con vosotros en la realización de este trabajo de Fin de Grado.

Darío, nunca me cansaré de agradecerte todo lo que has hecho por mí. Recuerdo como si fuese ahora aquel verano de 2019 cuando acudí a ti en busca de ayuda y motivación para seguir adelante con el grado. Finalmente, lo hemos conseguido.

Agradezco también a Unai Arronategui sus enseñanzas y la oportunidad que me dio para comenzar en el mundo laboral. Me gustaría también dar las gracias a todos los profesores del Departamento de Informática e Ingeniería de Sistemas, por transmitirme sus conocimientos durante estos años y por haberme formado tanto profesional como personalmente.

No puedo dejar de mencionar a mis compañeros de carrera, en especial a José Navarro, Raquel Roy y Sergio Hernández, con quienes he compartido horas de estudio, frustraciones y alguna que otra discusión, pero sobretodo momentos inolvidables. Qué suerte haberlo podido compartir con vosotros! Gracias por vuestro apoyo y ánimo en los momentos más difíciles. Esto también es vuestro.

A mi amigo Alberto Soro, gracias por estar siempre sin pedir nada a cambio. A Joaquín Riva e Isma, gracias por cuidarme y guiarme por el buen camino desde el primer día. A David Espeja por enseñarme unos valores que me siguen acompañando a día de hoy. A Mariser y Mariano, gracias por ser casa.

A todas aquellas personas y amigos que, de alguna manera, han contribuido a que esté donde estoy hoy, gracias.

A mi familia, mi madre y mi hermana, os debo eterno agradecimiento. Gracias mamá por no dejarme caer y por todo lo que has luchado por nosotros. Jamás podré recompensarlo. A mi hermana, gracias por estar siempre dispuesta a sacarnos una sonrisa, incluso en los peores momentos. Sois mi pilar fundamental.

Por último, me gustaría acordarme de mi padre. Gracias también a ti papá. Estoy seguro de que allá donde estés, estarás sacando pecho orgulloso.

Lo hemos conseguido. Gracias una vez más.

Estudio de Entornos de Ejecución Confiables sobre la Arquitectura RISC-V

RESUMEN

En la actualidad, y debido al aumento de ataques informáticos realizados en los últimos tiempos, mantener a salvo la información almacenada en cualquier tipo de dispositivo electrónico es uno de los principales objetivos de cualquier usuario o compañía. Una de las posibles soluciones es crear un entorno aislado en el cual, aunque el sistema operativo haya sido comprometido, los datos sensibles estarán protegidos.

Se denomina *Trusted Execution Environment* a aquella zona de ejecución segura dentro de un procesador. Es un área donde el código ejecutado y los datos accedidos están protegidos y aislados gracias a políticas de privacidad y permisos de acceso, garantizando así, que ningún usuario que no tenga los permisos necesarios pueda acceder, evitando modificaciones en el código y en su comportamiento.

El concepto de *Trusted Execution Environment* surge por primera vez en 2009 en uno de los documentos publicados por la OMTP Open Mobile Terminal Platform[1], un foro creado entre las grandes compañías para discutir acerca de la estandarización de determinados elementos de la tecnología móvil.

Dicho concepto, consiste en proveer un mecanismo de aislamiento a nivel hardware además de un sistema operativo seguro ejecutado por encima de dicho nivel de aislamiento. Sin embargo, el término se ha generalizado para referirse a una zona de ejecución segura dentro de un procesador.

Hoy en día, la mayoría de los dispositivos electrónicos que se encuentran a nuestro alrededor, utilizan *TEEs*. Ordenadores, smartphones, videoconsolas, y hasta dispositivos como una Smart Tv, hacen uso de dichos entornos.

Índice

Capítulo 1

Introducción y objetivos

Hoy en día, la información es el activo más valioso para individuos y organizaciones. Por ello, la seguridad de los datos es esencial para mantener la privacidad, la confidencialidad y la integridad de la información. La mayoría de los sistemas informáticos almacenan información crítica que debe ser protegida de accesos no autorizados. En este contexto, los secretos informáticos, claves privadas, contraseñas, certificados, credenciales de autenticación, incluso datos médicos, son especialmente importantes para mantener la seguridad de la información. [1]

Con el aumento constante de los dispositivos conectados a internet y la creciente cantidad de datos sensibles que se manejan, la necesidad de proteger la privacidad y autenticidad de los usuarios se ha vuelto una prioridad cada vez más apremiante.

En este sentido, existen diferentes técnicas y herramientas que pueden ayudar a proteger los secretos informáticos de accesos no autorizados. El uso de algoritmos criptográficos robustos y la implementación de medidas de seguridad adecuadas son esenciales para garantizar la protección de la información crítica.

Además, el uso de tecnologías como los módulos de seguridad de hardware (*TPM, Trusted Platform Modules*) pueden proporcionar un mayor nivel de protección ya que están diseñados para proteger la información confidencial mediante la implementación de medidas de seguridad avanzadas, como la autenticación de hardware y la encriptación de datos. Por otro lado, los entornos de ejecución seguros (*Trusted Execution Environmentes, TEE*) son herramientas que combinan las capacidades de los TPM y del software para proteger la información crítica y reducir el riesgo de ataques externos. Estos entornos proporcionan un mayor nivel de seguridad para las aplicaciones y datos sensibles, al limitar el acceso de los usuarios no autorizados y prevenir la ejecución de código malicioso.

Sin embargo, es importante destacar que ninguna medida de seguridad es completamente infalible y la seguridad de la información siempre es un proceso en constante evolución, es por ello que merece la pena resaltar una de las bases de la

seguridad informática, el principio de mínimo privilegio [2].

La seguridad informática es un conjunto de medidas, técnicas y herramientas que se utilizan para proteger los sistemas y la información de posibles amenazas y ataques externos o internos. En otras palabras, se trata de garantizar la privacidad, integridad y disponibilidad de los recursos informáticos. Su importancia ha ido creciendo con los años y como curiosidad el micro-kernel seL4, que se emplea en los entornos de ejecución segura de los dispositivos Apple, ha recibido el premio ACM Software System 2022 [3].

1.1. Motivación

En la era digital actual, la seguridad de la información y la protección de los secretos se han convertido en prioridades fundamentales para individuos y organizaciones. Mantener datos confidenciales a salvo de amenazas internas y externas es un desafío constante, especialmente cuando se trata de información delicada, como pueden ser las contraseñas o los números de tarjeta de crédito de los usuarios. Para abordar esta necesidad, es esencial contar con soluciones sólidas que permitan confinar secretos de manera segura, evitando el acceso no autorizado y protegiendo los datos sensibles.

Para proveer un nivel alto de seguridad, se requiere una combinación de varios elementos clave. En primer lugar, es fundamental contar con un entorno seguro donde se puedan almacenar los secretos de manera confidencial. En segundo lugar, dicho entorno debe ser capaz de proteger la información incluso en situaciones en las que un atacante tenga acceso físico al hardware que la contiene. En este sentido, la arquitectura de la solución debe ser resistente a intentos de robo de información o manipulación maliciosa.

Por ejemplo, en sistemas operativos basados en Unix y Linux, el usuario *root* es un usuario con privilegios de administrador o superusuario. El usuario *root* tiene acceso completo y sin restricciones al sistema y puede realizar cualquier tarea, incluyendo la instalación de software, la modificación de configuraciones de sistema, la creación y eliminación de usuarios y la gestión de permisos de archivos y directorios. En resumen, el usuario *root* tiene el control absoluto del sistema y es capaz de realizar operaciones críticas y cambiar la configuración del sistema en su totalidad.

Sin embargo, este acceso ilimitado también implica un riesgo significativo para la seguridad del sistema. La cuenta de usuario *root* se convierte en un punto único de fallo, lo que significa que si la cuenta cae en manos equivocadas, el sistema completo puede estar en peligro.

Una de las preocupaciones comunes en la seguridad de la información es la posibilidad de que un usuario malintencionado con acceso privilegiado, como un usuario *root*, pueda robar los números de tarjeta de crédito u otros secretos cuando se almacenan

en texto plano.

Aunque hay mecanismos para evitar que esto suceda, tales como la autenticación de dos factores y la implementación de políticas de seguridad estrictas, siempre existe el riesgo de que alguien pueda obtener acceso a la cuenta de *root*. Es por eso, que es crucial que se implementen medidas de seguridad adicionales para proteger la cuenta, incluyendo la implementación de contraseñas seguras y la limitación del acceso a la cuenta solo a aquellos que necesitan usarla. Para un nivel de seguridad superior, se pueden emplear entornos de ejecución segura, lo que eliminaría este problema.

1.2. Objetivos

El objetivo principal de este TFG es comprender qué es un entorno de ejecución segura (*TEE*), cómo funciona y experimentar con él sobre una arquitectura abierta.

- Analizar y comprender el concepto de entorno de ejecución segura (*TEE*), sus componentes y su evolución histórica.
- Estudiar las distintas aproximaciones a los entornos de ejecución seguros y la herramientas necesarias para su uso, funcionamiento y puesta en ejecución.
- Poner en marcha un entorno de emulación que permita experimentar con *TEEs*.
- Desarrollar un aplicación para la generación aleatoria de *nonces*.
- Comprender los mecanismos que ofrece la arquitectura RISC-V para la gestión de TEEs.

1.3. Alcance

Cabe destacar que al inicio del proyecto no se conocía nada sobre un entorno de ejecución segura, sobre la arquitectura abierta RISC-V o sobre cómo emular dicha arquitectura utilizando *QEMU* [4]. Es por ello que se destaca aquí el trabajo de investigación y documentación realizado por parte del autor, así como el trabajo de puesta en marcha del entorno.

Una vez comprendido el funcionamiento de la arquitectura, y tras poner en marcha el entorno sobre el cual se va a trabajar, se ha sido capaz de ejecutar un pequeño programa de ejemplo. Para ello, ha sido necesario comprender todos y cada uno de los componentes que entran en ejecución desde la compilación de los ficheros fuente y los distintos módulos, hasta la ejecución del programa en cuestión dentro del enclave aislado.

A partir de aquí, se ha desarrollado una aplicación para la generación aleatoria de *nonces* en un entorno de ejecución segura.

1.4. Estructura de este documento

El presente Trabajo Fin de Grado consta de seis capítulos. En el Capítulo 1 se introduce el problema analizado y se explica la motivación y objetivos del proyecto. En el Capítulo 2 se plantea el background tecnológico, evolución histórica y una aproximación al estado del arte; en el Capítulo 3, por su parte, se describe la arquitectura RISC-V. Tras plantear la arquitectura y los mecanismos que ofrece, en el Capítulo 4, se profundiza sobre la plataforma de desarrollo utilizada. En base a la información recogida en los capítulos anteriores, en el Capítulo 5, se presenta la validación experimental realizada mediante la prueba de concepto. Finalmente, en el Capítulo 6, se presentan las conclusiones de este trabajo.

Como complemento, se incluyen cuatro anexos de carácter técnico. El Anexo A explica con más detalle el funcionamiento del *PMP* y como se compone cada una de sus entradas; en el Anexo B se presentan los principios de diseño de Keystone; en el Anexo C, se profundiza sobre la conexión al cluster ATPS; en el Anexo D se explica la configuración aplicada para poner en funcionamiento el entorno de trabajo; finalmente, en el Anexo E se explican las modificaciones realizadas en el código fuente.

Capítulo 2

Estado del Arte

En este capítulo, se va profundizar sobre los entornos de ejecución segura, su evolución histórica y alguna de las implementaciones comerciales más comunes. Históricamente, la mayoría de los procesadores no incluían ningún mecanismo de seguridad pero a lo largo de los últimos 30 años han ido añadiendo características para hacerlos más seguros. Originalmente, algunos sistemas basados en arquitecturas como AMD64 tendieron hacia el uso de chips externos como los *Trusted Platform Modules* para realizar las operaciones de seguridad, mientras que los sistemas basados en procesadores ARM para móviles aprovecharon las ventajas en cuanto a integración de chips de los System-on-Chip [5] para añadir *Trusted Execution Environments* y proveer mecanismos de seguridad. En años recientes, fabricantes de procesadores como Intel mediante *Software Guard Extensions* (SGX) han añadido también extensiones de seguridad como parte de su arquitectura.

2.1. Trusted Platform Module

Durante casi dos décadas, los fabricantes de hardware han confiado en los Módulos de Plataforma Confiables (TPMs) para una computación segura. En la figura 2.1 se puede apreciar un ejemplo de Trusted Platform Module (TPM), un chip de seguridad integrado en la placa base de los equipos que permite la creación de un entorno seguro y aislado para la ejecución de procesos críticos y la protección de datos sensibles. Este chip se encarga de garantizar la seguridad de la plataforma [6].

El TPM debe permanecer separado físicamente del sistema al que informa: el sistema anfitrión. Puede ser un sólo componente físico que se comunica con el sistema anfitrión a través de un bus simple. Los TPM contienen módulos que permiten la generación de números aleatorios, generación de claves de criptograficas, notificación en caso de cambios de energía por parte del host y almacenamiento persistente entre otras características, tal y como se aprecia en la figura 2.2 [7].

Figura 2.1: Trusted Platform Module listo para ser conectado en una placa madre

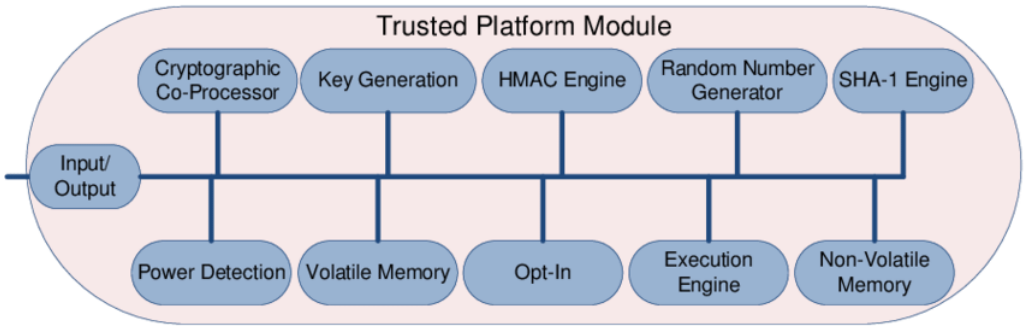

Figura 2.2: Arquitectura genérica de un TPM. [7]

Un TPM es utilizado en diversos casos de uso como puede ser la encriptación de discos, autenticación de usuarios, protección de firmware, seguridad de inicio seguro o gestión de derechos digitales (DRM) [8]. En general, es utilizado en cualquier aplicación que requiera una seguridad sólida y confiable.

Algunas de las implementaciones comerciales dónde se utiliza TPM son BitLocker [9] o Lenovo ThinkPad [10], utilizadas para proteger los datos del disco duro y garantizar la seguridad del sistema, incluyendo protección contra manipulaciones físicas (intentos de extracción o modificación del chip), autenticación de usuarios o acceso remoto. Además en 2009, la especificación de Trusted Computing Group (TCG) para un TPM fue ratificada como un estándar ISO [11].

2.2. Trusted Execution Environment.

Los *Trusted Execution Environments* son entornos que permiten la ejecución de manera aislada al resto del sistema fuera del sistema operativo principal e independientemente del resto de procesos del computador, que fueron definidos en la Open Mobile Terminal Platform y ratificados en 2009 [12]. En estos entornos aislados se puede garantizar la autenticidad del código ejecutado, la integridad de los estados en tiempo de ejecución y la confidencialidad de su código, datos y estados de ejecución almacenados en una memoria persistente. Además, deben ser capaces de proporcionar una atestación remota que demuestre su confiabilidad para terceros. El contenido del *TEE* no es estático ya que se puede actualizar de manera segura. El *TEE* es capaz de

resistir los ataques de software, así como los ataques físicos realizados en la memoria principal del sistema [13].

Los conceptos utilizados en un *TPM* como almacenamiento seguro y memoria aislada se expanden en un *TEE* con conceptos como E/S aislada y protegida y también, RAM aislada.

Los *TEEs* se han convertido en una parte importante de la arquitectura de seguridad de los dispositivos, más particularmente en los teléfonos móviles, ya que los usuarios confían cada vez más en estos dispositivos para almacenar y procesar información confidencial como claves criptográficas, contraseñas o huellas biométricas. Además la seguridad de estos sistemas es un tema crítico debido a la gran cantidad de datos personales que manejan.

2.2.1. Evolución histórica

Los primeros teléfonos móviles con *TEE* basados en hardware aparecieron hace casi dos décadas en forma de teléfonos Nokia que utilizaban procesadores de Texas Instruments [14].

Alrededor de 2003, ARM propuso desarrollar un aislamiento de hardware en todo el sistema para la ejecución segura para Nokia. La cooperación entre ARM y Texas Instruments tenía objetivos empresariales independientes separados de las necesidades de Nokia, pero también estaba el propio interés de dicha compañía. Esto proporcionó a Nokia la posibilidad de implementar un entorno seguro en cualquier chip que implemente la arquitectura de seguridad de ARM, lo que más tarde se conocería como ARM TrustZone [15].

Cabe destacar que ARM Trustzone es una tecnología de seguridad de hardware basada en procesadores ARM y está disponible bajo una licencia comercial de ARM, lo que implica que puede ser utilizada por cualquier fabricante de dispositivos que elija un procesador ARM que lo soporte.

Esta solución, divide los recursos del sistema en una zona segura y una zona normal, con recursos que no se comparten entre estas zonas. Su seguridad depende en gran medida del procesador a utilizar y de cómo el fabricante del dispositivo implementa la tecnología. Sus principales usos residen desde smartphones hasta servidores de alto rendimiento. En la figura 2.3 se pueden diferenciar dos stack diferentes, en el lado izquierdo la aplicación de usuario no segura (*untrusted*), y en el lado derecho la aplicación segura (*trusted*) en el mundo aislado.

Tal y como se ha comentado anteriormente, tras definirse y ratificarse el término *TEE* en 2009 por la OMTP [12], el término fue desarrollado y estandarizado por GlobalPlatform en 2011. GlobalPlatform describe una arquitectura de sistema para

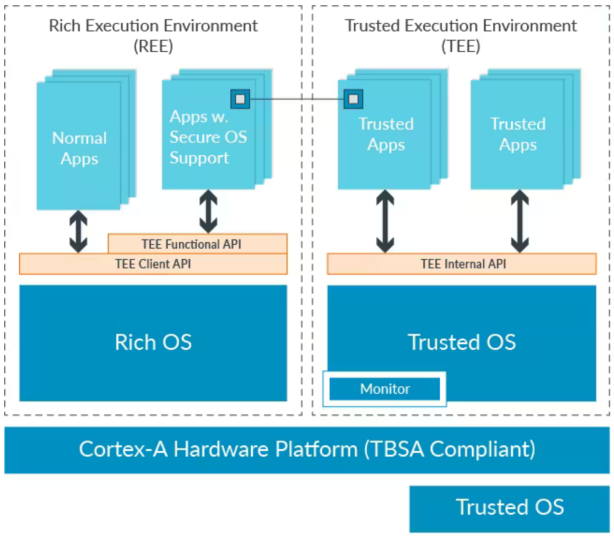

Figura 2.3: Componentes ARM Trustzone. [16]

implementar entornos de ejecución segura para aplicaciones de seguridad en dispositivos móviles y otros dispositivos embebidos [17]. Además, estandarizó el TEE API y describió que el *TEE* ofrece protección de contenidos digitales, autenticación de seguridad-código crítico (pagos digitales) e integridad del código del sistema (firmware). De esta manera, se favoreció la implementación de un *TEE* Open Source, evitando así el problema de licencias y propietarios.

A raíz de dicha especificación surgen proyectos como OP-TEE [18] tal y como se aprecia en la figura 2.4, una implementación basada en ARM Trustzone siguiendo estándar de Trusted Execution Environment de GlobalPlatform. Fue desarrollado por la Fundación Linaro.

Apple Secure Enclave es una tecnología de seguridad de hardware que se utiliza en dispositivos Apple y también fue creada a partir de ARM Trustzone. Apple Secure Enclave fue lanzado en septiembre de 2013 con el iPhone 5s, cuyo diseño está patentado y es propiedad exclusiva de Apple, por lo que sólo está presente en sus dispositivos. Dicha tecnología se encuentra en los System-on-Chip de Apple y no está separado físicamente del resto del procesador principal. En lugar de ser un procesador independiente en un chip externo, el Secure Enclave es un coprocesador integrado dentro del mismo chip que contiene el procesador principal, con su propio arranque seguro y aislamiento personalizado para proteger los datos y las operaciones que maneja. Apple tiene un control total sobre el diseño y la implementación de Secure Enclave, y también proporciona funcionalidades adicionales como la protección de datos biométricos. [19]

Siguiendo la figura 2.4, en 2015 surge una tecnología denominada *Intel SGX* [20],

enfocada a la creación de entornos de ejecución seguros en los procesadores Intel, proporcionando a los desarrolladores capacidad para aislar código y datos sensibles de manera confidencial, garantizando la integridad de estos incluso cuando son ejecutados por software privilegiado en el mismo sistema.

Aunque la mayoría de los *TEE* de código abierto se han desarrollado para la arquitectura ARM, hay algunos proyectos que han diseñado *TEE* para otras arquitecturas.

El proyecto Sancus [21], se centra en la seguridad en sistemas embebidos, garantizando dicha seguridad sin confiar en ningún software de infraestructura en el dispositivo. El único hardware es el *TCB* (Trusted Computing Base). Además su costo en hardware es bajo.

Por último, Keystone [22] es un proyecto de código abierto que proporciona un entorno de ejecución seguro para aplicaciones en sistemas basados en la arquitectura RISC-V. Keystone TEE se ejecuta en hardware RISC-V y proporciona aislamiento de hardware y software para aplicaciones de seguridad. Se va a profundizar en esta herramienta en los siguientes capítulos ya que este TFG está basado en la arquitectura RISC-V y en Keystone como marco de trabajo.

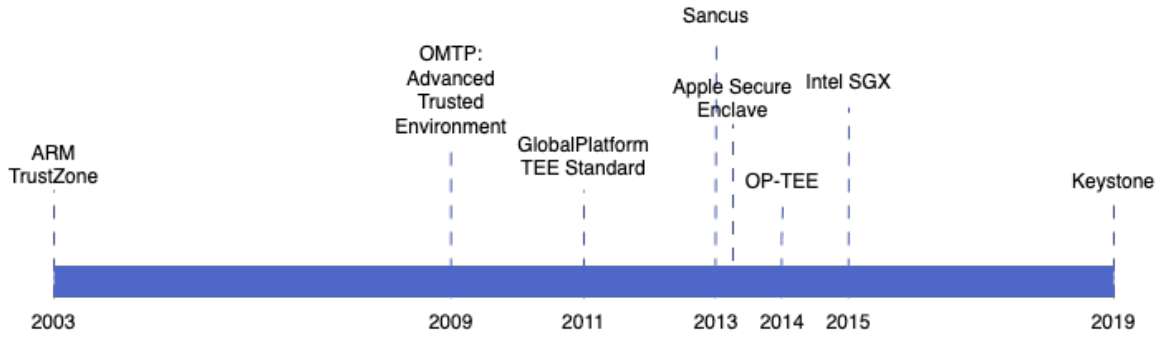

Figura 2.4: Evolución temporal de los *TEE*.

Capítulo 3

RISC-V

En este capítulo se va a describir la arquitectura RISC-V y las características más importantes que ofrece, tales como descripción del conjunto de instrucciones, modos de privilegio, root of trust, Phyisical Memory Protection y Supervisor Binary Interface de cara a implementar TEEs.

3.1. Arquitectura RISC-V

La arquitectura RISC-V [23] es una arquitectura con un conjunto de instrucciones reducido (*RISC*, Reduced Instruction Set Computing), abierto y libre de regalías que se ha convertido en una opción popular para el diseño de procesadores. La arquitectura se comenzó a desarrollar en 2010 en la Universidad de California, Berkeley y ahora es mantenida por la Fundación RISC-V, una organización sin ánimo de lucro.

RISC, como concepto, se refiere a una filosofía de diseño de arquitectura de computadores que enfatiza la eficiencia en la ejecución rápida de un conjunto reducido de instrucciones simples, en contraste con las arquitecturas *CISC* (Complex Instruction Set Computing), que tienen un conjunto de instrucciones más extenso y de mayor funcionalidad.

En un sistema tradicional basado en Intel x86 o ARM, la micro-arquitectura y la arquitectura suelen ser controladas por la misma entidad empresarial. Lo cual ha unido de manera artificial niveles de abstracción que en un sistema informático deberían ser independientes. Esto se ha utilizado como una estrategia industrial para dificultar el acceso de nuevos fabricantes de chips pudiendo aprovechar arquitecturas existentes y ampliamente utilizadas.

En contraste, RISC-V solo especifica una arquitectura y la deja libre. Permite a los usuarios definir su propia micro-arquitectura segura, personalizada según las necesidades específicas de su aplicación o sistema. Esto brinda una mayor flexibilidad y adaptabilidad en el diseño de la micro-arquitectura, ya que no se limita a un conjunto

de instrucciones o características predefinidas.

La capacidad de personalizar dicha micro-arquitectura en RISC-V ofrece una serie de beneficios. Gracias a la modularidad, los usuarios pueden optimizar el rendimiento y la eficiencia de la unidad de ejecución segura al adaptarla a los requisitos de la aplicación. También pueden incorporar características específicas de seguridad según sea necesario, como mecanismos de detección de ataques o protocolos de cifrado personalizados. Esto permite una amplia variedad de implementaciones de la arquitectura, desde microcontroladores hasta servidores de alta gama.

A medida que las empresas buscan reducir costos y aumentar la flexibilidad, la adopción de la arquitectura RISC-V se ha acelerado. A día de hoy, la arquitectura RISC-V se ha vuelto más madura y ha demostrado su eficacia en diferentes aplicaciones, ganando tracción en la industria. Grandes empresas como Western Digital [24] y SiFive [25] están invirtiendo en la arquitectura y están desarrollando productos basados en ella. Esto unido al reconocimiento de dicha arquitectura por parte del Gobierno de España en el Programa Estratégico de Apoyo a la Industria de Semiconductores [26], genera una inercia y plantea un escenario en el que la arquitectura RISC-V podría ser una alternativa muy viable en un futuro.

3.1.1. Conjunto de instrucciones en RISC-V.

La arquitectura RISC-V [27] incluye varios conjuntos de instrucciones (ISA) que se dividen en una base y extensiones opcionales.

En el conjunto de instrucciones base se aprecian dos variantes: *RV32I* y *RV64I*. El número que aparece junto al nombre hace referencia a la longitud de las direcciones de memoria que el conjunto de instrucciones puede manejar (32 o 64 bits), e "I" significa "Integer", indicando que estas instrucciones realizan operaciones con números enteros. Ambos conjuntos de instrucciones (RV32I y RV64I), incluyen instrucciones para operaciones enteras, control de flujo, carga y almacenamiento, y manipulación directa de bits.

Además de estos conjuntos de instrucciones base, se pueden añadir extensiones opcionales para proporcionar funcionalidades extra. Este diseño modular permite a los diseñadores de chips adaptar la ISA a sus necesidades específicas, seleccionando las extensiones que necesitan y omitiendo las que no, favoreciendo así la eficiencia.

3.1.2. Modos de privilegio.

Los modos de privilegio en una arquitectura se refieren a los diferentes niveles de acceso y control que un programa o proceso tiene sobre el sistema. A veces también se

denominan niveles o anillos de protección. En cada nivel, un programa puede ejecutar distintas instrucciones, realizar ciertas operaciones y acceder a un conjunto de recursos hardware.

El propósito principal de los modos de privilegio es mejorar la seguridad y la estabilidad del sistema. Al limitar la actividad de los programas (incluyendo el sistema operativo), los modos de privilegio ayudan a prevenir que los programas de usuario interfieran con los recursos críticos del sistema o con otros programas. También pueden evitar que un programa malicioso o defectuoso cause daño al sistema.

A diferencia de la arquitectura x86 que utiliza una estructura de cuatro niveles (del anillo 0 al anillo 3), RISC-V [28] utiliza una estructura de tres niveles: modo usuario, modo supervisor y modo máquina; tal y como se aprecia en la figura 3.1, que van a ser descritos a continuación.

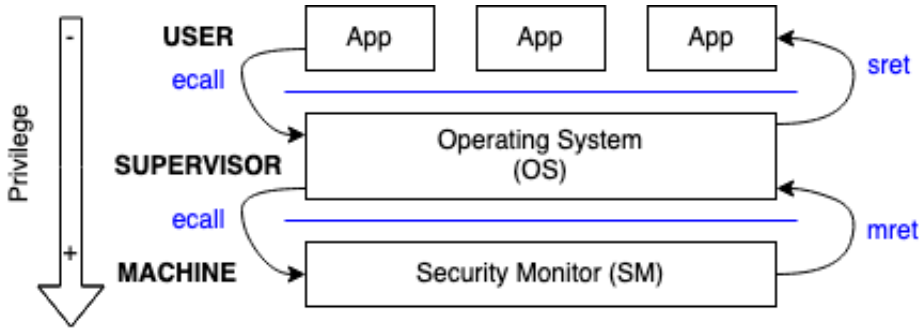

Figura 3.1: Niveles de privilegio de la arquitectura RISC-V. En color azul, instrucciones en lenguaje ensamblador necesarias para gestionar los cambios entre niveles de privilegio.

- *Modo usuario*: nivel más bajo de privilegio. Las aplicaciones de software de usuario se ejecutan en este modo. Las aplicaciones no pueden acceder directamente a los recursos de hardware ni realizar acciones que puedan afectar el sistema en su conjunto. Deben solicitar estos servicios al sistema operativo, que opera en el siguiente nivel de privilegio. Esto proporciona una importante capa de seguridad, ya que limita lo que las aplicaciones pueden hacer.

- *Modo supervisor*: destinado a los sistemas operativos, proporciona acceso adicional a las instrucciones y registros del procesador en comparación con el modo usuario. El software en el modo supervisor tiene la capacidad de controlar el acceso a los recursos del sistema y realizar llamadas al sistema utilizando la interfaz binaria del supervisor (*SBI*). Este modo es utilizado por el sistema operativo para realizar operaciones de bajo nivel, como la gestión de memoria y la gestión de interrupciones. Tanto el modo usuario como el modo supervisor pueden realizar llamadas al siguiente nivel de privilegio correspondiente.

– *Modo máquina*: modo de privilegio más alto en RISC-V. El software en el modo máquina tiene acceso completo a todos los recursos del sistema, incluyendo los registros del procesador, la memoria y los dispositivos de entrada/salida. Este modo es utilizado por el hipervisor o el firmware de arranque del sistema para configurar el hardware y realizar tareas de mantenimiento del sistema.

Además de estos tres modos, RISC-V también define modos adicionales para el uso de máquinas virtuales como el modo hipervisor (H-mode), que están fuera del ámbito de este TFG.

Para entender la comunicación entre los distintos modos de privilegio comentados anteriormente (usuario, supervisor y máquina), es necesario profundizar sobre algunos conceptos. Los cambios de contexto entre modos de privilegio se producen a través de interrupciones. Para ello, hay que ser capaces de generar una solicitud de interrupción y un retorno de esta. Por lo tanto, RISC-V ofrece instrucciones, que gestionan a nivel de software estas interrupciones de cambio de modo de privilegio:

– *Environment Call (ecall)*: instrucción para solicitar un cambio de modo de privilegio. Permite realizar peticiones desde un nivel de privilegio más bajo (usuario) para ejecutar código privilegiado en un nivel superior (kernel/supervisor), como las llamadas al sistema tal y como se aprecia en la figura 3.1. Por el contrario y como se muestra también en dicha figura, en otras ocasiones es el propio kernel (supervisor) el que se encuentra en modo de privilegio inferior y podría invocar al nivel superior (máquina) con una instrucción de este tipo.

– *mret*: permite realizar un retorno desde el modo de privilegio *máquina* al nivel de privilegio anterior, el modo supervisor.

– *sret*: instrucción utilizada para regresar al modo de privilegio anterior después de manejar una excepción o una interrupción en un programa en modo *supervisor*.

3.1.3. Root of Trust.

Root of trust o raíz de confianza es un concepto de seguridad que hace referencia a un mecanismo utilizado para establecer y garantizar un punto de partida seguro en el sistema a partir del cual se certifica la seguridad del resto de módulos y componentes del sistema.

En términos simples, el "root of trust" es un componente de hardware cuyo fabricante garantiza que es seguro y que no ha sido comprometido. Dicho componente hardware contiene unas claves criptográficas insertadas por el fabricante y actúa como

una autoridad certificadora (CA) en los procesos de firma digital. Este hardware se encarga de validar el siguiente componente software como podría ser el sistema de arranque, y una vez se ha validado correctamente dicho sistema de arranque es capaz de validar nuevos módulos como podría ser el SO, para continuar después con los drivers y así sucesivamente ... De esta manera se establece la cadena de confianza comentada previamente.

El objetivo del root of trust en RISC-V es dar un punto de partida seguro al Security Monitor (*SM*), código que se ejecuta en el modo máquina, a través de una concatenación de comprobaciones iniciada desde el hardware.

3.1.4. Pyhisical Memory Protection (PMP).

El PMP (Physical Memory Protection) en la arquitectura RISC-V [28] describe un mecanismo hardware que permite definir que regiones del espacio de direccionamiento son visibles durante la ejecución de una aplicación. La modificación de esta visibilidad sólo es posible en modo de privilegio máquina, y por lo tanto condiciona el espacio de direccionamiento visible de los modos supervisor y usuario.

Para definir la visibilidad del espacio de direccionamiento, la arquitectura define un banco de registros de control con un número máximo de 16 entradas. Por defecto, Keystone setea este valor a 8 entradas. Cada entrada de PMP contiene una dirección de memoria y una serie de atributos que especifican los derechos de acceso (lectura, escritura y ejecución) y si el acceso está permitido o no en los diferentes modos de privilegio.

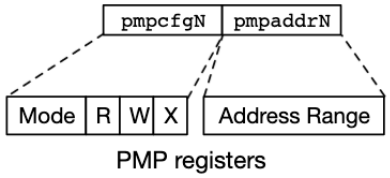

Figura 3.2: Ejemplo de entrada PMP en RISC-V.

Las entradas PMP actúan como una *whitelist* tal y como se aprecia en la figura 3.3, lo que significa que la memoria es inaccesible si no se definen ninguna de las entradas PMP. Las entradas PMP también tienen prioridad estática, de modo que la entrada PMP de número más bajo que coincida con cualquier byte de un acceso a memoria, además del modo de privilegio, determina si el acceso a memoria tiene éxito o falla. Si la solicitud de acceso coincide con una entrada de PMP y cumple con los derechos de acceso especificados, entonces se permite el acceso. Si no hay ninguna entrada de PMP que coincida o si la solicitud de acceso viola los derechos de acceso, entonces se produce una excepción. Una vez se ha iniciado el sistema, dichas entradas no pueden ser

modificadas hasta que la máquina se resetea, añadiendo así un nivel más de seguridad. Dicho funcionamiento se describe con mayor detalle en el anexo A.

Figura 3.3: Visibilidad de entradas PMP en Keystone.

3.1.5. Interfaces de comunicación.

Esta sección aborda dos aspectos fundamentales para el funcionamiento y la compatibilidad de software en la arquitectura RISC-V: el *ABI* (Application Binary Interface) y el *SBI* (Supervisor Binary Interface). Ambos son aspectos críticos de la arquitectura RISC-V que aseguran la cohesión y la compatibilidad entre diferentes componentes del software y el hardware. Permiten definir interfaces para que el programador pueda invocar a la función de cambio de modo de privilegio sin tener que preocuparse de cómo hacerlo.

El *ABI* es un conjunto de reglas y convenciones que establece la interfaz de comunicación entre el software de nivel de aplicación y el hardware subyacente. Define cómo los programas representan y pasan argumentos, gestionan pilas de llamadas, y acceden a los registros de la CPU, entre otros aspectos. Al proporcionar directrices uniformes para la interacción entre distintos componentes del programa, el *ABI* garantiza la portabilidad y la compatibilidad del código compilado con diversas implementaciones de hardware y sistemas operativos.

Por otro lado, el *SBI* se enfoca en la comunicación entre el software de nivel supervisor (como el sistema operativo) y la parte privilegiada del hardware, conocida como modo máquina. El *SBI* ofrece una serie de llamadas de funciones o servicios que permiten al software de nivel supervisor acceder a recursos específicos de hardware y realizar operaciones de privilegio, sin comprometer la seguridad y el aislamiento del sistema. Esta interfaz es especialmente útil en escenarios con múltiples niveles de privilegios, garantizando la gestión eficiente de interrupciones, la configuración del manejo de memoria y otros aspectos cruciales para el funcionamiento del sistema. El *SBI* define una convención de llamada de función y un a estructura de argumentos

de llamada de sistema facilitando así, la comunicación entre los distintos modos de privilegio.

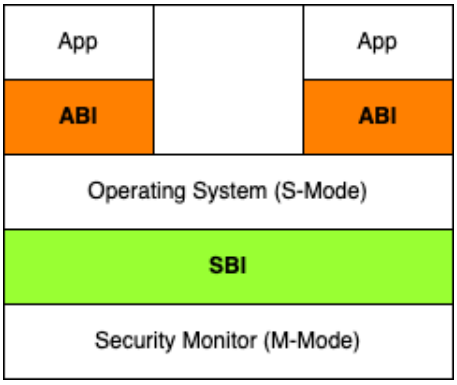

Figura 3.4: ABI y SBI en RISC-V.

Tal y como se aprecia en la figura 3.4, cada aplicación se comunica a través de una *ABI* con el sistema operativo. Así como las aplicaciones interactúan con el sistema operativo a través de una *ABI*, los sistemas operativos RISC-V interactúan con un entorno de ejecución de mayor privilegio (*SM*) a través de una interfaz binaria de supervisor (*SBI*). Un *SBI* comprende el ISA de nivel de usuario y nivel de supervisor junto con un conjunto de llamadas de funciones de *SBI*.

Capítulo 4

Keystone

Keystone es una plataforma de desarrollo de código abierto para la construcción de entornos de ejecución segura [22]. Está basado en la arquitectura RISC-V y ofrece un entorno de desarrollo que incluye un compilador, un simulador y una biblioteca de software para el desarrollo de enclaves seguros.

Pero ... ¿qué es un enclave? Hasta ahora se ha tratado el concepto de entorno de ejecución segura o *TEE*. Un enclave en Keystone es una entidad de software que proporciona un entorno aislado y seguro para la ejecución de código, utilizando los mecanismos de ejecución que ofrece la arquitectura para soportar *TEE*s. Keystone se va a apoyar en las bases vistas en los capítulos anteriores para proporcionar un espacio protegido (*enclave*) dentro de un sistema más amplio (*TEE*) donde se pueden ejecutar aplicaciones y servicios sensibles sin preocuparse de que puedan ser comprometidos o accedidos por partes no autorizadas. Dichas bases se recogen en los principios de diseño de Keystone, definidos en el anexo B.

Este capítulo describe el funcionamiento de Keystone [29], sus componentes y arquitectura, centrando el foco en dos principales eventos, el proceso de arranque, y el proceso de creación y ejecución de un enclave. Para ello ha sido necesario realizar un estudio previo de las partes de dicho marco de trabajo, gracias a la documentación pública existente, y un exhaustivo proceso de investigación a través de documentación oficial, artículos científicos, ficheros fuente, pruebas de concepto y una correcta puesta en funcionamiento del entorno.

4.1. Componentes

Antes de describir los distintos componentes, es necesario tener una imagen completa del sistema que se va a describir, comprendiendo así, las dos zonas que entran en escena. Para ello, y siguiendo la figura 4.1, se aprecia en el lado izquierdo y marcada en color rojo la zona *untrusted* compuesta por la(s) aplicación(es) de usuario(s) y el

sistema operativo. Cabe destacar, que se habla de zona *untrusted* porque la aplicación de usuario se ejecuta simultáneamente con otras aplicaciones o procesos. En dicha zona, corren todas las aplicaciones con sus utilidades convencionales y puesto que son vulnerables, se consideran *untrusted*. Por el contrario, en el lado derecho de la figura y en color azul, aparece la zona *trusted, isolated*, un lugar donde se pueden almacenar los secretos relativos a cada aplicación de manera aislada a la zona *untrusted*. El aislamiento del que se habla en esta última zona, es garantizado por parte del *SM*, que limitará el intercambio de información entre las dos zonas.

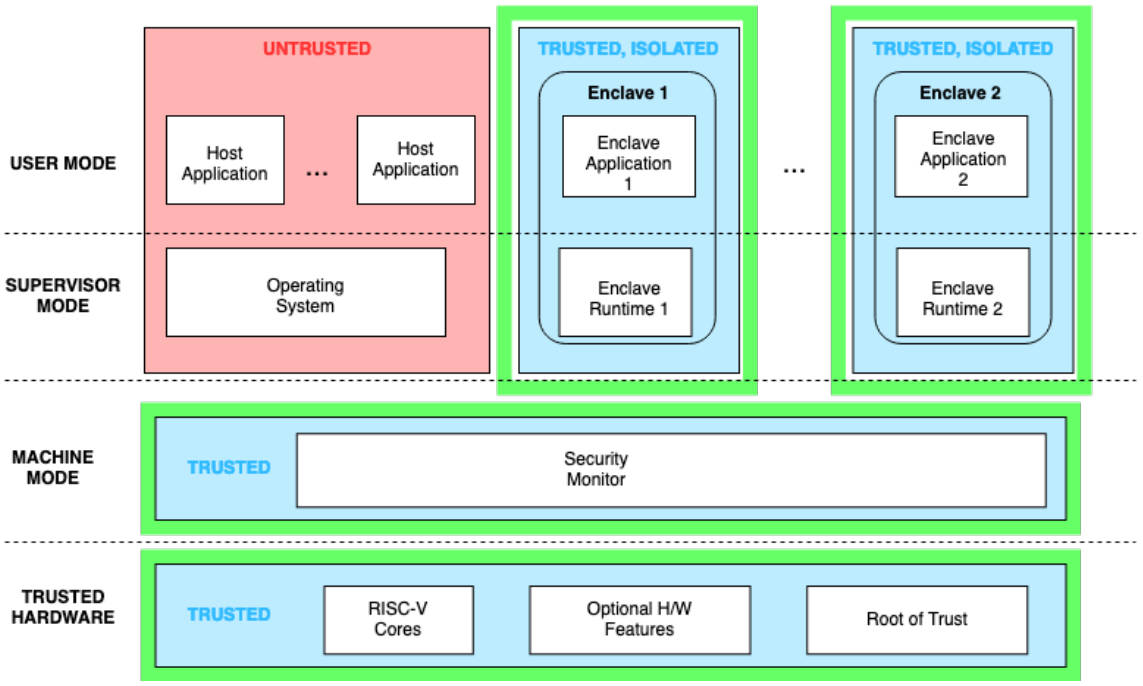

Figura 4.1: Esquema de ejecución con entorno seguro donde conviven aplicaciones confiables *trusted* y no confiables *untrusted* en el mismo sistema.

Se detallan a continuación los distintos componentes (ver figura 4.1). Cabe recordar que un *enclave* hace referencia al componente software que una aplicación *untrusted* utiliza para ejecutarse en zona *trusted* donde preserva sus secretos. Dicho enclave será no persistente durante toda su vida útil, es decir, será creado, ejecutado y destruido. El concepto enclave con el que se va a trabajar está formado por la *Enclave Application* y el *Enclave Runtime*.

– *Trusted Hardware*: paquete de CPU construido por un vendedor de confianza, debe contener al menos un núcleo compatible con RISC-V permitiendo el uso del PMP y el root of trust. El hardware también puede contener características extra como partición de cache, encriptación de memoria, generación criptográfica de números aleatorios...

– *Security Monitor (SM)*: software que es ejecutado en el nivel de privilegio máquina con una pequeña Trusted Computing Base o *TCB* (parte de un sistema informático que contiene todos los elementos responsables de la política de seguridad). Proporciona una interfaz para controlar el ciclo de vida del enclave así como para utilizar características específicas de la plataforma (hardware). Gestiona el aislamiento y los límites entre los enclaves y el sistema operativo.

– *Enclave application (eapp)*: aplicación que se ejecuta en la zona *trusted* con nivel de privilegio más bajo (usuario). Dicha aplicación está compuesta por un código específico que se ejecuta de manera aislada para evitar exponer secretos.

– *Runtime (RT)*: software que es ejecutado en modo supervisor *trusted* y que da soporte a las *eapp* haciendo las funciones de una biblioteca con los servicios mínimos para que ésta pueda ejecutarse correctamente. Incluye la implementación de funcionalidades como llamadas al sistema, manejo de interrupciones, gestión de la memoria virtual...

4.2. Proceso de arranque del sistema

Este apartado describe de manera simplificada el proceso de arranque para comprender como interactúan entre sí los distintos componentes durante este proceso.

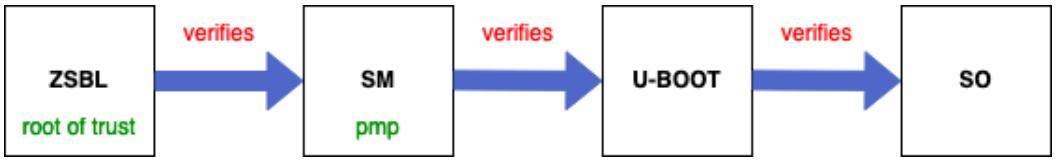

Figura 4.2: Proceso de arranque del sistema y elementos que intervienen en él. En color verde elementos hardware [30].

Siguiendo la figura 4.2, el ciclo de vida comienza desde el hardware, haciendo uso del Zero Stage Bootloader (*ZSBL*) el chip del procesador comienza un proceso denominado *chain of trust* o cadena de confianza encargado de validar cada uno de los componentes, tal y como se ha comentado previamente.

Mediante procedimientos criptográficos, dicha cadena de confianza, permite verificar el *SM* en cada inicio de la máquina, permitiendo así asegurar, que no ha sido alterado. Como se aprecia en la figura 4.3, cuando el sistema arranca, dicho *SM* asigna la primera entrada del *PMP* (PMP[0]) para aislar su propia memoria, evitando que nadie que no sea el propio *SM* pueda acceder a su espacio de memoria.

Después, el *SM* asigna la última entrada del *PMP* (PMP[N]) al *SO*, cubriendo así el resto de la memoria. En este momento, gracias al Universal Boot Loader (*U-Boot*) [31],

se carga el *SO* en memoria, y se le pasa el control. De esta manera, el *SO* arranca teniendo visible toda la memoria excepto la correspondiente al *SM*.

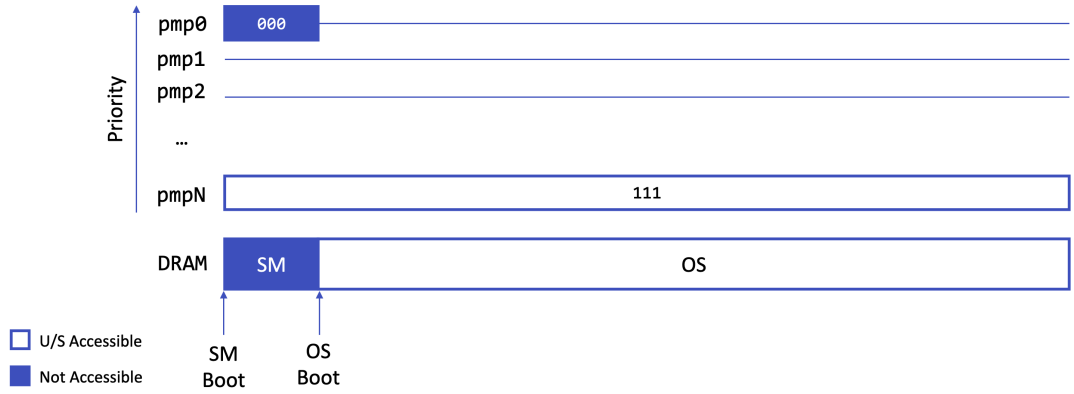

Figura 4.3: SM y PMP en proceso de arranque del sistema.

A partir de aquí, el usuario es capaz de desarrollar una aplicación para ser ejecutada en un enclave si lo considera necesario. Cabe resaltar que un arranque seguro es necesario para poder crear y lanzar un enclave, pero no obliga a ello.

4.3. Ciclo de vida del enclave

Para comprender mejor el ciclo de vida de un enclave, se presentan a continuación los diferentes estados y siguiendo la figura 4.4, se aprecian las transiciones de un estado a otro. Dichas transiciones han sido modificadas por el autor de este TFG para conseguir enclaves persistentes, tal y como se explicará en el capítulo de validación experimental.

- *ALLOCATED*: Estado del enclave una vez le ha sido asignado un identificador.
- *FRESH*: Estado del enclave una vez ha sido validado su hash.
- *RUNNING*: Estado del enclave una vez se pone en ejecución.
- *STOPPED*: Estado del enclave una vez se detiene.
- *DESTROYING*: Estado del enclave una vez ha sido detenido y ha recibido la orden de destruirse.

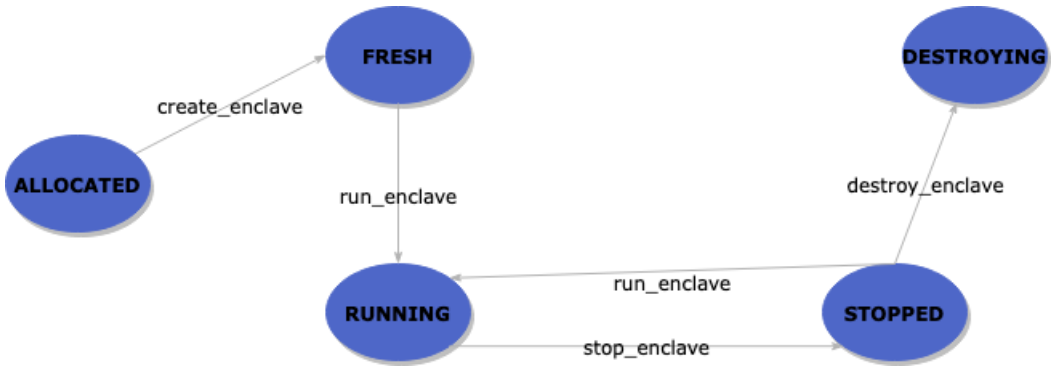

Figura 4.4: Ciclo de vida de un enclave.

4.4. Creación del enclave

Keystone da soporte para desarrollar y generar todos los binarios que forman parte de una aplicación segura, los generados desde los ficheros fuente relativos al host (*untrusted*), a la enclave application y al runtime (*trusted*). Estos incluyen todos los procesos de comunicación entre dicho host a nivel de usuario (*untrusted*), el sistema operativo y el modo máquina, así como la comunicación entre el modo máquina, el *RT* y la *eapp* (*trusted*).

Para la creación del enclave, es el usuario mediante la host application (*untrusted*) el encargado de crearlo. Una vez se ejecuta el método que proporciona la API para crear un enclave, se ejecuta una instrucción de tipo *ecall* para cambiar el nivel de privilegio a supervisor. Una vez alcanzado este nivel, el SO reserva una zona de memoria y la inicializa con la tabla de páginas del enclave y la aplicación tal y como se aprecia en el paso 1 de la figura 4.5.

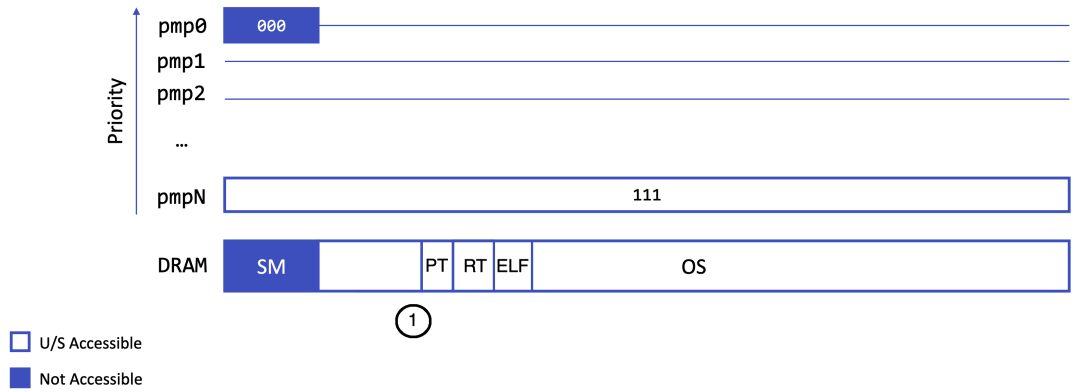

Figura 4.5: SM y PMP en proceso de creación del enclave.

En este momento, el *SO* podría escribir en esa zona de memoria, lo que se vería reflejado después en el proceso de verificación. Después, y tras elevar de nuevo el

nivel de privilegio ejecutando de nuevo la instrucción *ecall*, el *SM* inicializa la entrada correspondiente al enclave en el *PMP* tal y como se aprecia en el paso 2 de la figura 4.6, comprueba el hash de dicho enclave (si el *SO* ha modificado el contenido de esas direcciones, la verificación no tendrá éxito) y en caso de que dicho hash coincida, cambia los bits de permiso de dicha zona de memoria al valor 000 (lectura, escritura y ejecución) de manera que no será visible al *SO*. El *SO* puede pedir al *SM* la creación de tantos enclaves como entradas libres haya en el *PMP*. Llegados a este punto, podría solicitar la creación de hasta seis enclaves más ya que PMP[0] corresponde al *SM* y PMP[1] pertenece a este nuevo enclave.

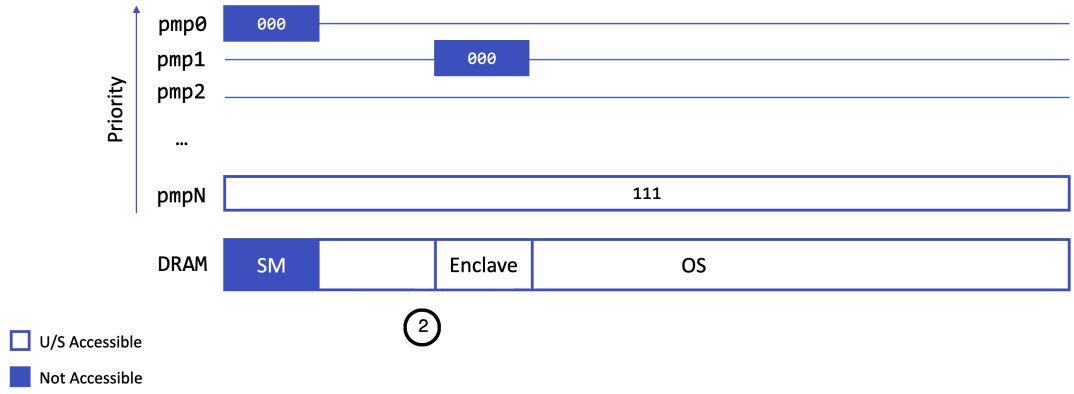

Figura 4.6: SM y PMP en proceso de creación del enclave.

4.5. Ejecución de un enclave

Para la ejecución de un enclave, es el usuario mediante la host application (*untrusted*) el encargado de lanzarlo. Para ello, se disparará una instrucción de tipo *ecall*, encargada de cambiar el modo de privilegio. Una vez alcanzado el modo máquina, el *SM* comprueba si existe una entrada en el *PMP* para dicho enclave, y siguiendo la figura 4.7, se encarga de cambiar los permisos haciendo visible la zona correspondiente al enclave que se va a ejecutar, escribiendo el valor 111 (lectura, escritura y ejecución respectivamente). Dicho enclave será capaz de acceder únicamente a la zona de memoria asignada para si mismo. Tal y como se aprecia en la figura 4.1, el enclave en proceso de ejecución, estaría encuadrado en la parte derecha, en la zona *trusted*, interactuando con la *eapp* a nivel de usuario, el *runtime* a nivel de supervisor y el *SM* en modo máquina.

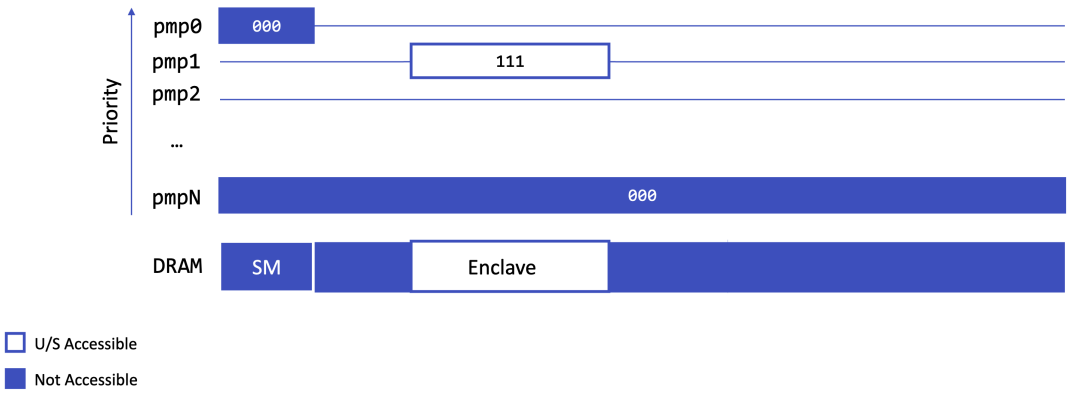

Figura 4.7: SM y PMP en proceso de ejecución de un enclave.

4.6. Destrucción del enclave

El proceso de destrucción de un enclave implica la liberación de la memoria correspondiente a dicho enclave, eliminar su entrada correspondiente en el *PMP* y devolver el control al *SO*, realizando el cambio de la zona visible de la memoria. Todo ello realizado por parte del SM. Para ello, el host *trusted* ha tenido que transferir el control al *RT* mediante una *ecall*, y este *RT* al *SM* que se ejecuta en nivel de privilegio máquina de la misma manera. El estado de la memoria tras destruir un enclave queda tal y como refleja la figura 4.8. En dicha figura se aprecia la liberación del espacio de memoria del enclave y la liberación de la entrada correspondiente en el *PMP*. Esto implica que una vez ha sido destruido un enclave, no se va a poder acceder más a su espacio de memoria, aunque el *SM* coloque un nuevo enclave en la misma entrada del *PMP*.

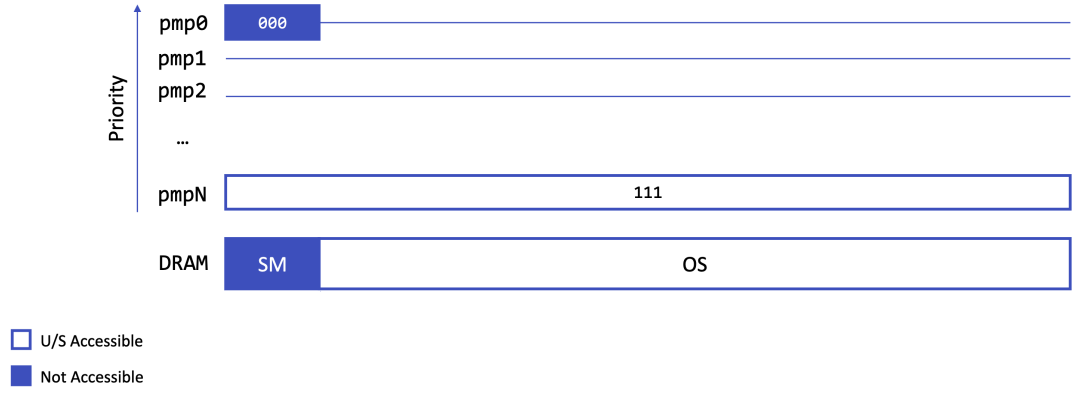

Figura 4.8: SM y PMP tras destruir un enclave.

4.7. Buffer de memoria compartida

Para compartir información entre ambas zonas, *untrusted* y *trusted* se emplea un buffer de memoria compartida. Dicho buffer permite la comunicación desde la zona *trusted*, a la zona *untrusted*. Es decir, permite enviar información al exterior del enclave, pero no al interior. Es una comunicación en un sólo sentido.

Para ello, el *SO* es capaz de colocar un buffer de memoria compartida en su espacio de memoria, el *SM* usa la última entrada del *PMP*, PMP[N] para permitir visibilidad de dicho buffer al enclave durante su ejecución tal y como se puede apreciar en la figura 4.9.

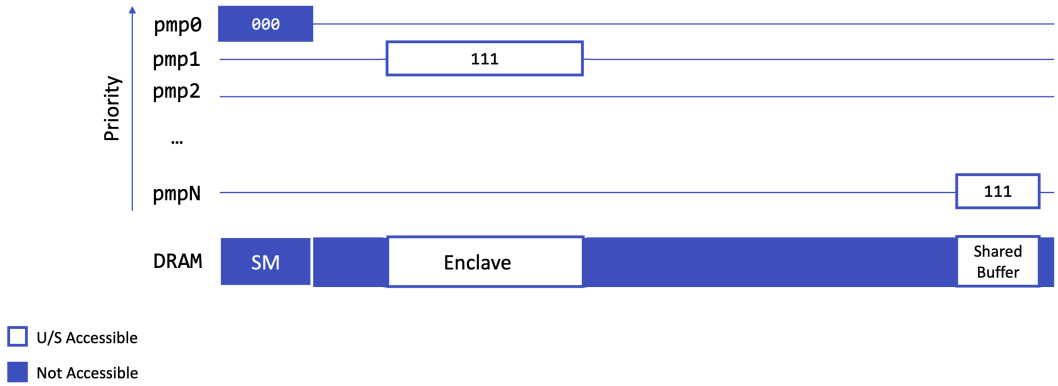

Figura 4.9: SM y PMP con enclave y buffer de memoria compartida.

Su funcionamiento está basado en una serialización de la información a enviar desde el enclave, y un conjunto de funciones encargadas de gestionar el buffer y des-serializar dicha información en el host *untrusted*. Dichas funciones son conocidas como *edge calls* y su funcionamiento se explica en detalle en el anexo E.3.

Capítulo 5

Validación experimental

Este capítulo describe el caso de uso diseñado para realizar una nueva prueba de concepto sobre Keystone, generación de *nonces* pseudo-aleatorios dentro de un enclave, para validar experimentalmente sus capacidades.

5.1. Descripción

La prueba de concepto a realizar es la generación de *nonces* pseudo-aleatorios [32]. Un nonce es es un número arbitrario que se puede usar una única vez en una comunicación criptográfica. Los nonces suelen ser cadenas numéricas y son empleados en diversas aplicaciones para asegurar la autenticación y la seguridad.

5.2. Entorno de trabajo

Aunque la arquitectura RISC-V ofrece mecanismos hardware opcionales para la generación de números aleatorios, este TFG se ha realizado utilizando *Keystone* en *QEMU*, tal y como se describe en el anexo D.

El esquema completo de trabajo con todas las máquinas involucradas y las distintas capas del sistema se puede apreciar en la figura 5.1.

5.3. Metodología

La validación experimental se ha realizado sobre el cluster ATPS. El cluster ATPS es una infraestructura de computación de altas prestaciones del grupo de Arquitectura de Computadores de la Universidad de Zaragoza (gaZ) [33]. ATPS se usa principalmente para simular modelos funcionales y temporales a nivel micro-arquitectura (procesadores, caches y redes de interconexión). Este TFG se ha realizado utilizando el nodo 004 de dicho cluster. En el anexo C se dan más detalles de cómo se ha realizado la conexión a dicho cluster.

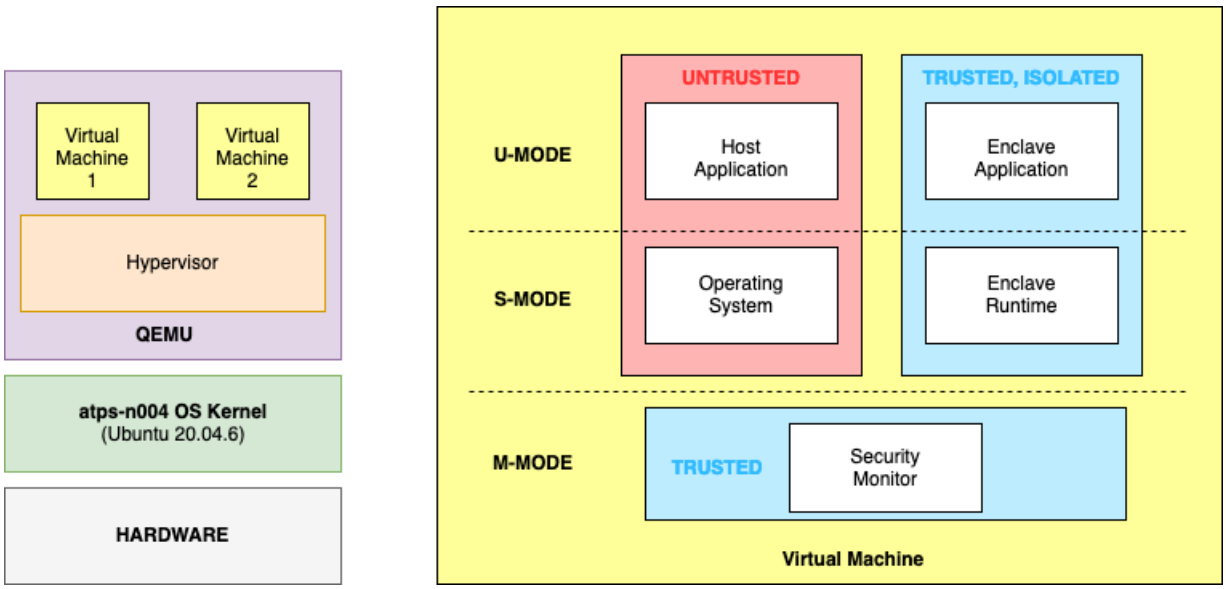

Figura 5.1: Capas y componentes del sistema completo

5.4. Prueba de concepto

Tal y como se ha comentado previamente, un *enclave* está compuesto por una *eapp* (Enclave Application) y un *runtime* (Enclave Runtime). Además de esto, necesita de la existencia de un host *untrusted* que será el encargado de inicializar y lanzar el enclave.

Siguiendo la figura 5.2, se pretende dar una aproximación simplificada de como se realiza la comunicación de ida, desde el host *untrusted* para inicializar un enclave, crearlo y ponerlo en ejecución, pasándole el control para que sea capaz de ejecutar el código relativo al generador de nonces desde la propia *eapp*.

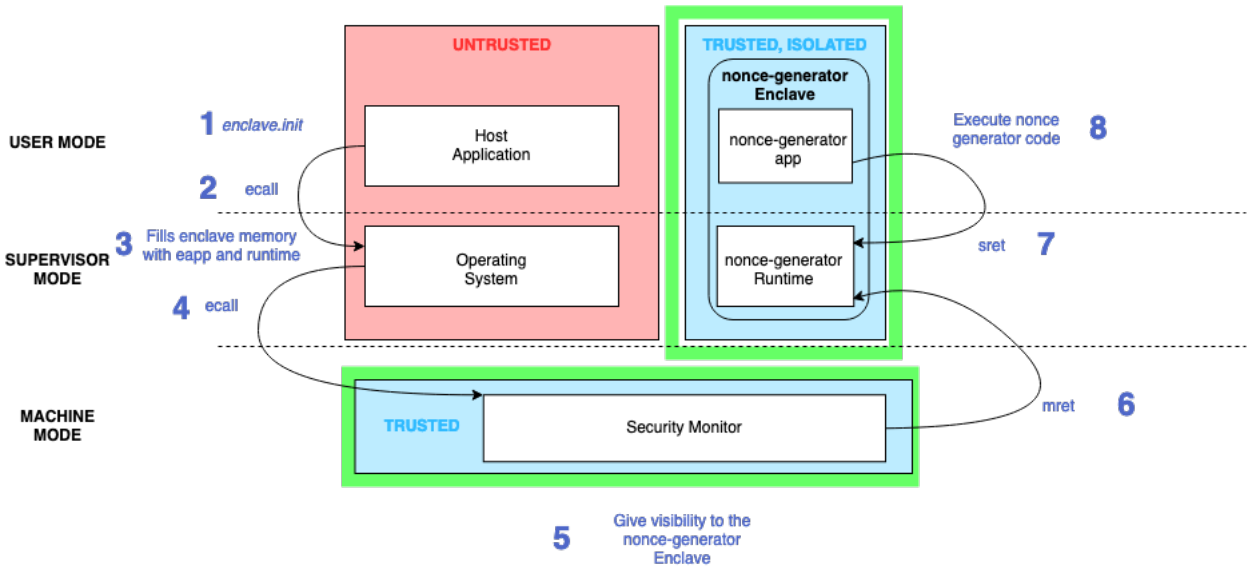

Figura 5.2: Flujo de operaciones realizadas para pasar el control desde host *untrusted* a *eapp*.

Para ello, el host *untrusted* en nivel de privilegio usuario ejecuta una instrucción de inicialización del enclave (1), que acabará ejecutando una *ecall* para solicitar un cambio de modo de privilegio a supervisor, dándole el control al *SO* (2). Dicho SO, se encarga de reservar una zona de memoria e inicializarla con la tabla de páginas del enclave y la aplicación (3) y (4), tal y como se ha explicado previamente en la sección 4.4.

Una vez se ha alcanzado el nivel de privilegio máquina, el *SM* será el encargado de dar visibilidad a las zonas de memoria de dicho enclave (5) y mediante una instrucción de tipo *mret*, devolverá el control al *RT* en modo de privilegio supervisor (6). Dicho *RT* acabará ejecutando una instrucción *sret* (7), para transferirle el control a la *eapp*, en modo usuario, que será quien ejecute el código relativo al generador de nonces (8).

Con el objetivo de comprender cómo se realiza el proceso de comunicación de vuelta, desde el enclave al host *untrusted*, (utilizando el buffer de memoria compartida), se ha decidido utilizar una estructura de datos de tipo *struct* con varios campos en su interior. La estructura de datos utilizada se puede muestra a continuación en el listing 5.1. Dicho proceso de comunicación se explica más en detalle en el anexo E.3, ya que es simétrico a la comunicación de ida explicada en esta sección.

Listing 5.1: Estructura utilizada para la comunicación entre *eapp* y *host untrusted*

```
typedef struct Cont {
        char array0[16];
        char array1[16];
        char array2[16];
        char array3[16];
        char array4[16];
        char array5[16];
        unsigned short number0;
} Cont;
```

Esta decisión se ha tomado tras observar y comprender, el ejemplo proporcionado por los desarrolladores de Keystone, en el que se imprime por salida estándar la cadena "Hello World", siendo ésta enviada desde la *eapp* al host *untrusted* haciendo uso de las *ocalls* y del buffer de memoria compartida.

Inicialmente se intentó llevar al límite el ejemplo proporcionado por los desarrolladores, es por eso, que se decidió utilizar una estructura de datos *struct* con varios campos en su interior. Tras definir el formato de dicha estructura de datos para la serialización y des-serialización, se fue consciente de que el código fuente original solamente soportaba *ocalls* con un máximo de cinco parámetros. Lo que limitaba el número de parámetros a compartir entre la *eapp* y el host *untrusted*.

Esto permitió implementar una nueva *ocall*, de nombre *ocall_two* con siete parámetros tal y como se muestra en la figura E.5, para comprobar que se había

comprendido el comportamiento correctamente.

Llegados a este punto, se comenzó a trabajar en la prueba de concepto definitiva. Para ello, ha sido necesario generar un *nonce* en la *eapp*. El código fuente de la *eapp* y del *host untrusted* se puede consultar en el anexo E.2.

Una vez se ha sido capaz de generar un *nonce*, hay que enviar esa información de vuelta al host *untrusted* e imprimirlo por salida estándar, lo que se explica en detalle en el anexo E.3, tal y como se ha referenciado previamente.

Después, ha sido necesario completar el siguiente desafío. Cabe recordar, que el funcionamiento original de Keystone, no soporta enclaves persistentes. Esto implica que un enclave es creado, ejecuta el código necesario para generar un nonce y compartirlo con el host *untrusted* y se destruye. Para ello, ha sido necesario modificar el ciclo de vida del enclave en Keystone, en el código fuente relativo al *SM*, concretamente ampliando la lógica a partir de la cual un enclave puede ser puesto en ejecución, que contempla el estado *FRESH* contemplado inicialmente, e incorpora el estado *STOPPED*, para que una vez ha sido detenido un enclave, pueda volver a lanzarse. Los diferentes estados del ciclo de vida del enclave se pueden apreciar en la figura 4.4 y las modificaciones realizadas en el código se muestran en la figura 5.3.

```
unsigned long run_enclave(struct sbi_trap_regs *regs, enclave_id eid)
{
  int runable;

  spin_lock(&encl_lock);
  runable = (ENCLAVE_EXISTS(eid)
            && (enclaves[eid].state == FRESH || enclaves[eid].state == STOPPED));
  if(runable) {
    enclaves[eid].state = RUNNING;
    enclaves[eid].n_thread++;
    //sbi_printf("I'm in RUNNING state from RUN. \n");
  }
  spin_unlock(&encl_lock);

  if(!runable) {
    return SBI_ERR_SM_ENCLAVE_NOT_FRESH;
  }

  // Enclave is OK to run, context switch to it
  context_switch_to_enclave(regs, eid, 1);

  return SBI_ERR_SM_ENCLAVE_SUCCESS;
}
```

Figura 5.3: Código fuente *run-enclave* modificado. Encuadrado en color rojo se puede apreciar el nuevo estado STOPPED añadido a la lógica.

El resultado de la prueba de concepto devuelve por salida estándar desde el host *untrusted* tres valores de nonces generados consecutivamente por el enclave, tal y como se puede apreciar en la siguiente figura 5.4.

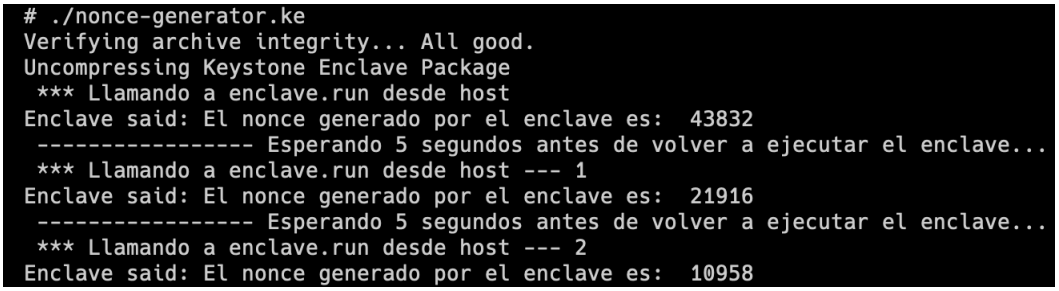

Figura 5.4: Resultado de la ejecución del generador de nonces.

Capítulo 6

Conclusiones

Los entornos de ejecución seguros (*TEE*) proporcionan un mayor nivel de seguridad para las aplicaciones y datos sensibles, al limitar el acceso de los usuarios no autorizados y prevenir la ejecución de código malicioso.

Además, este trabajo de fin de grado ha permitido asentar las bases obtenidas durante las asignaturas del grado que incluyen conceptos relacionados con los niveles de privilegio de una arquitectura, llamadas al sistema y proceso de arranque.

También ha proporcionado al estudiante una visión completa de las distintas soluciones existentes en el mercado y un conocimiento de la solución con la arquitectura RISC-V, cumpliendo así uno de los objetivos más prioritarios, ya que antes de comenzar este trabajo de Fin de Grado no se conocía nada sobre *TEE*s y RISC-V.

Antes de valorar las conclusiones obtenidas, cabe resaltar de nuevo que ninguna solución de seguridad es segura al completo y que los *TEE* añaden un nivel más de seguridad pero no garantizan que un sistema no pueda ser comprometido.

Respecto a la tecnología utilizada, no es una tecnología sencilla de utilizar, ha requerido un exhaustivo trabajo de comprensión y documentación previo, así como puesta en marcha del entorno de trabajo. Como posible mejora se puede destacar una mejor documentación o una definición de las APIs más sencilla. Aunque se incluye un foro de Google para compartir dudas, no es suficiente. Dicha tecnología no es completa en su totalidad ya que como se ha podido comprobar, no soporta transferencia de información desde la zona *untrusted*, a la zona *trusted*, al interior del enclave.

Respecto al trabajo realizado, se ha podido comprobar que Keystone ofrece *enclaves* no persistentes, y gracias al trabajo realizado por parte del autor, se ha conseguido que dichos enclaves sean persistentes. Permitiendo así, que una vez ha sido creado un enclave, sea posible volverlo a ejecutar. Esto, unido a la generación de nonces en su interior, ha permitido completar con éxito la prueba de concepto planteada como uno de los objetivos iniciales.

Capítulo 7

Bibliografía

[1] Jefatura del Estado. Ley orgánica 3/2018, de 5 de diciembre, de protección de datos personales y garantía de los derechos digitales, 2018.

[2] Swarup Bhunia and Mark Tehranipoor. *Hardware Security*. Morgan Kaufmann, 2019.

[3] Association for Computing Machinery. Acm software system award, 2022.

[4] Qemu. `https://www.qemu.org/`. Accessed: [29/07/2023].

[5] V.S. Chakravarthi and S.R. Koteshwar. *System on Chip (SOC) Architecture: A Practical Approach*. Springer Nature Switzerland, 2023.

[6] Justin D. Osborn and David C. Challener. Trusted platform module evolution, 2013.

[7] Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. *An Introduction to the Trusted Platform Module and Mobile Trusted Module*, pages 71–93. Springer, 2014.

[8] Qiong Liu, Reihaneh Safavi-Naini, and Nicholas Sheppard. Digital rights management for content distribution. volume 21, pages 49–58, 01 2003.

[9] Microsoft Inc. Introducción a bitlocker, 2023.

[10] Lenovo. A technical introduction to the use of trusted platform module 2.0 with linux, 2017.

[11] ISO. Iso/iec 11889-1:2015, 2015.

[12] Open Mobile Terminal Platform. Advanced trusted environment document, 2009.

[13] Mohammed Achemlal Mohamed Sabt and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not, 2015.

[14] Jerome Azema and Gilles Fayad. Mobile security technology: making wireless secure, 2008.

[15] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security, 2004.

[16] ARM. Development of tee and secure monitor code - arm trustzone.

[17] Global Platform. The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market, 2015.

[18] Linaro. Open portable trusted execution environment.

[19] Inc Apple. Apple platform security guide, 2021.

[20] Intel Corporation. Innovative instructions and software model for isolated execution.

[21] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 479–498, Washington, D.C., August 2013. USENIX Association.

[22] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. Association for Computing Machinery, 2020.

[23] RISC-V Foundation. Risc-v.

[24] Western Digital Corporation. Western digital delivers new innovations to drive open standard interfaces and risc-v processor development. 2018.

[25] SiFive Inc. The investment heard around the world. 2022.

[26] Gobierno de España. Perte: Chip microelectrónica y semiconductores.

[27] RISC-V Foundation. The risc-v instruction set manual, unprivileged specification version.

[28] RISC-V Foundation. The risc-v instruction set manual, privileged specification version.

[29] Keystone Enclave. Keystone source code release, 2019.

[30] Atish Patra and Anup Patel. Risc-v summit 2019: An introduction to risc v boot flow.

[31] Behan Webster. Introduction to the embedded boot loader u-boot.

[32] Phillip Rogaway. Nonce-based symmetric encryption. In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2004.

[33] Computer Architecture Group of the University of Zaragoza. Computer architecture group of the university of zaragoza webpage.

[34] W3Techs. Usage statistics of content languages for websites, 2017. Last accessed 16 September 2017.

[35] Cmake. `https://cmake.org/`. Accessed: [28/07/2023].

[36] Repositorio github con código fuente keystone. `https://github.com/keystone-enclave/keystone`.

[37] Foro keystone en google groups. `https://groups.google.com/g/keystone-enclave-forum`.

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Funcionamiento PMP

Cuando un programa intenta acceder a una dirección de memoria, el hardware de RISC-V verifica la solicitud contra las entradas de *PMP*, mediante el uso de un conjunto de registros de estado de control (CSR) en RISC-V. Cada núcleo puede tener de 0 a 16 registros *PMP*, cada uno de los cuales consta de una configuración (*pmpcfg*) y un registro de dirección (*pmpaddr*) para definir una entrada PMP. El registro *pmpcfg* define el modo de direccionamiento y los bits de permiso, y *pmpaddr* especifica el rango de direcciones codificando la dirección usando un modo de direccionamiento seleccionado. Hay tres modos de direccionamiento:

- *NA4*: palabra alineada de 4 bytes.
- *NAPOT*: alineado naturalmente en potencias de dos.
- *TOR*: tope de rango.

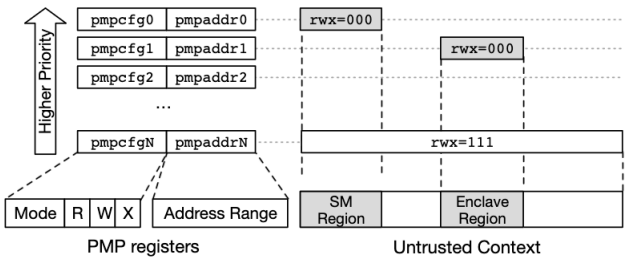

Figura A.1: PMP en Keystone.

El *PMP* proporciona una capa adicional de seguridad en los sistemas basados en RISC-V, ya que ayuda a evitar que los programas maliciosos o defectuosos accedan a regiones de memoria que no deberían o simplemente para aislar diferentes tareas o procesos entre sí. Sin embargo, requiere que el Security Monitor (*SM*) configure correctamente las entradas de *PMP* para proteger adecuadamente las regiones de memoria.

Anexos B

Principios de diseño Keystone

Para entender el funcionamiento de Keystone es necesario comprender los principios de diseño con los que ha sido construido.

- Utilizar capas programables y primitivas de aislamiento debajo del código no confiable.
  Se ha diseñado un Security Monitor (*SM*) para hacer cumplir las garantías de *TEE* en la plataforma, utilizando cuatro propiedades del modo máquina:
  - Programable por los proveedores de plataforma.
  - Cumple el principio de mínimo privilegio.
  - Controla la delegación de hardware de las interrupciones y excepciones en el sistema.
  - Permite el control del *PMP* de RISC-V para aislar las características de control de memoria asignada al hardware en tiempo de ejecución.
- Separa la gestión de recursos y la comprobación de seguridad en el diseño de *TEE*.
  El *SN* es responsable de hacer cumplir las políticas de seguridad con un código mínimo en el nivel de máximo privilegio, lo que reduce el *TCB* y permite una presentación clara de abstracciones. El *RT* y la *eapp* residen en el espacio de direcciones del enclave y están aislados del sistema operativo no confiable o de otras aplicaciones de usuario. El *RT* maneja el ciclo de vida del código del usuario que se ejecuta en el enclave, administra la memoria, brinda servicios de llamadas al sistema... Cada instancia de enclave puede elegir su propio *RT*, que nunca se comparte con otros enclaves. Para la comunicación con el *SM*, el *RT* utiliza un conjunto limitado de funciones API a través de la interfaz binaria del supervisor RISC-V (SBI), para salir o pausar el enclave y solicitar operaciones del *SM* en nombre del eapp.

– Diseño modular.
  Keystone utiliza la modularidad (*SM*, *RT*, *eapp*) para soportar una variedad de cargas de trabajo. Esto libera a los proveedores de plataformas Keystone y a los programadores de Keystone de tener que adaptar sus requisitos y aplicaciones heredadas a un diseño de *TEE* existente.

– Permite una mínima configuración del TCB, si es necesario.
  Keystone puede instanciar *TEE* con el *TCB* mínimo para casos de uso específicos. El programador del enclave puede optimizar aún más el *TCB* a través de la elección del *RT* y las bibliotecas *eapp* utilizando la separación de privilegios usuario/kernel existente. Por ejemplo, si el *eapp* no necesita soporte de libc o gestión dinámica de memoria, Keystone no los incluirá en el enclave.

Anexos C

Conexión al cluster ATPS

La conexión a dicho cluster se ha realizado utilizando el protocolo *secure shell*, conocido popularmente como *SSH*. Es un protocolo que tiene como función ofrecer acceso remoto a un servidor. La principal peculiaridad es que este acceso es seguro, ya que toda la información va cifrada. Esto evita que pueda filtrarse y que un tercero pueda ver esos datos. Dicha conexión se refleja en la figura C.1.

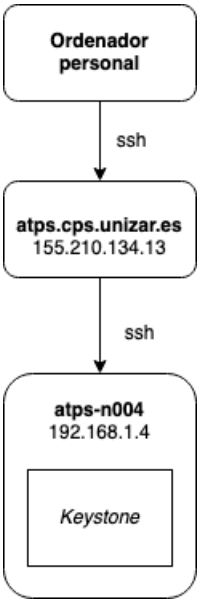

Figura C.1: Flujo de comunicación con el cluster ATPS y el nodo 004 mediante *ssh*.

Anexos D

Entorno de trabajo

QEMU es un software de código abierto que proporciona emulación y virtualización de sistemas y procesadores. Su principal objetivo es permitir la ejecución de programas y sistemas operativos diseñados para una arquitectura de hardware específica en una plataforma diferente, sin requerir modificaciones en el código fuente original. Es por ello, que al no disponer de hardware RISC-V que incluye entre sus características un módulo para la generación de números aleatorios, se ha decidido implementar un generador de *nonces* de manera software en el interior del enclave.

D.1. Instalación de dependencias

Se ha utilizado la versión *20.04.6* de la distribución Ubuntu como sistema operativo.

El primer paso, ha sido instalar todas las dependencias y paquetes necesarios para poder utilizar Keystone en el emulador *QEMU*, así como instalar la versión correspondiente de dicho software, *5.0.0*.

D.2. Descarga de fuentes pre-compilados de RISC-V.

El siguiente paso ha sido ejecutar el script *fast-setup.sh*, cuyo objetivo es descargar todas las herramientas pre-compiladas de *RISC-V* y extraerlas en el directorio */riscv64*, así como establecer los submódulos del proyecto (linux, buildroot y qemu). Dicho script instalará el *SDK* (Sofware Development Kit) de Kestyone, un conjunto de herramientas, bibliotecas, documentación y ejemplos, en el directorio */sdk/build64*.

D.3. Compilación de ficheros fuente

Para la compilación de los fuentes, *Keystone* hace uso de *CMake* [35]. CMake es un conjunto de herramientas *open source*, diseñadas para compilar, testear y

empaquetar software. Controla el proceso de compilación de software haciendo uso de una plataforma simple y archivos de configuración independientes del compilador, generando así ficheros *makefile* y espacios de trabajo nativos que se pueden usar en el entorno del compilador de su elección.

Tras invocar a la herramienta *CMake* mediante el comando *cmake*, se generarán los Makefile correspondiente que van a permitir compilar todos los fuentes del proyecto.

Previamente, ha sido necesario entender la generación de dichos Makefile mediante la herramienta *CMake* y los ficheros *CMakeList.txt*, así como añadir las opciones necesarias para compilar los fuentes de la prueba de concepto, el generador de nonces, tal y como se aprecia en la figura D.1.

```
set(eapp_bin nonce-generator)
set(eapp_src eapp/eapp_nonce.c)
set(host_bin nonce-generator-runner)
set(host_src host/host_nonce.cpp)
set(package_name "nonce-generator.ke")
set(package_script "./nonce-generator-runner nonce-generator eyrie-rt")

if(RISCV32)
  set(eyrie_plugins "rv32 freemem")
else()
  set(eyrie_plugins "freemem")
endif()

# eapp

add_executable(${eapp_bin} ${eapp_src})
target_link_libraries(${eapp_bin} "-nostdlib -static" ${KEYSTONE_LIB_EAPP} ${KEYSTONE_LIB_EDGE})

target_include_directories(${eapp_bin}
  PUBLIC ${KEYSTONE_SDK_DIR}/include/app
  PUBLIC ${KEYSTONE_SDK_DIR}/include/edge)

# host

add_executable(${host_bin} ${host_src})
target_link_libraries(${host_bin} ${KEYSTONE_LIB_HOST} ${KEYSTONE_LIB_EDGE})
# add -std=c++11 flag
set_target_properties(${host_bin}
  PROPERTIES CXX_STANDARD 11 CXX_STANDARD_REQUIRED YES CXX_EXTENSIONS NO
)
target_include_directories(${host_bin}
  PUBLIC ${KEYSTONE_SDK_DIR}/include/host
  PUBLIC ${KEYSTONE_SDK_DIR}/include/edge)

# add target for Eyrie runtime (see keystone.cmake)

set(eyrie_files_to_copy .options_log eyrie-rt)
add_eyrie_runtime(${eapp_bin}-eyrie
  "v1.0.0"
  ${eyrie_plugins}
  ${eyrie_files_to_copy})

# add target for packaging (see keystone.cmake)

add_keystone_package(${eapp_bin}-package
  ${package_name}
  ${package_script}
  ${eyrie_files_to_copy} ${eapp_bin} ${host_bin})

add_dependencies(${eapp_bin}-package ${eapp_bin}-eyrie)

# add package to the top-level target
add_dependencies(examples ${eapp_bin}-package)
```

Figura D.1: Ejemplo fichero CMakeList.txt

Tras compilar dichos ficheros Makefile generados por CMake, mediante el comando *make* y (*make* nonce-generator) para la nueva prueba de concepto, se genera como

resultado un binario denominado *nonce-generator.ke*. Dicho binario tendrá que ser copiado al interior de la máquina virtual y posteriormente será necesario regenerar la imagen de QEMU para que contenga dicho binario en su interior con el comando (*make* image).

Anexos E

Código fuente

El código fuente ha sido modificado a partir del código original, disponible en GitHub. [36]. También existe un foro en el que compartir ideas con otros desarrolladores disponible en Google Groups [37].

E.1. Principales APIs de Keystone

Para emplear Keystone y modificar su código, es necesario conocer sus principales interfaces de programación. Estas se dividen principalmente en tres grandes grupos:

- *APIs del host*: Se encuentra en el directorio *sdk/src/host* y contiene las operaciones utilizadas por el host *untrusted* para crear, destruir y comunicarse con los enclaves a través de la clase *Enclave*.

- *APIs del enclave*: Situada en *sdk/src/app*, utilizada por el código que se ejecuta dentro del enclave (*eapp*) para disparar la comunicación de vuelta con la aplicación del host *untrusted*, manejar la memoria, y realizar otras operaciones relacionadas con el enclave.

- *APIs de comunicación*: se puede encontrar bajo la ruta *sdk/src/edge* y proporciona los mecanismos para gestionar las *edge calls* por parte de ambos lados, host (untrusted) y eapp (trusted). Las *edge calls* son las funciones que cruzan de un lado al otro, desde la eapp (trusted) al host (untrusted), haciendo uso del buffer de memoria compartida. Keystone las denomina internamente *ocall*, outbound calls.

E.2. Host *untrusted* y eapp

E.2.1. Host *untrusted*

Se muestra a continuación el código fuente de la *eapp* en la figura E.1, dicho código se encuentra bajo la ruta */sdk/examples/nonce-generator/host/host_nonce.cpp*. En rojo, la instrucción que dispara la creación de un enclave. En verde, la instrucción que dispara su puesta en ejecución tras haber sido creado.

```
int
main(int argc, char** argv) {
  Keystone::Enclave enclave;
  Keystone::Params params;

  params.setFreeMemSize(1024 * 1024);
  params.setUntrustedMem(DEFAULT_UNTRUSTED_PTR, 1024 * 1024);

  enclave.init(argv[1], argv[2], params);

  enclave.registerOcallDispatch(incoming_call_dispatch);

  /* We must specifically register functions we want to export to the enclave. */
  register_call(OCALL_PRINT_STRUCT, print_string_wrapper);

  edge_call_init_internals(
      (uintptr_t)enclave.getSharedBuffer(), enclave.getSharedBufferSize());

  printf(" *** Llamando a enclave.run desde host \n");

  enclave.run();

  printf(" ----------------- Esperando 5 segundos antes de volver a ejecutar el enclave... \n");

  sleep(5);

  printf(" *** Llamando a enclave.run desde host --- 1 \n");

  enclave.run();

  printf(" ----------------- Esperando 5 segundos antes de volver a ejecutar el enclave... \n");

  sleep(5);

  printf(" *** Llamando a enclave.run desde host --- 2 \n");

  enclave.run();

  return 0;
}
```

Figura E.1: Código fuente host *untrusted*.

Los primeros pasos sirven para configurar el tamaño de la memoria del enclave y la dirección y tamaño del buffer de memoria compartida.Después, se inicializa la memoria del enclave con la *eapp* y el *runtime* pasados por parámetros en la función *enclave.init*.

Para gestionar las *edge calls*, el enclave debe registrar el método encargado de manejar dichas *edge calls*, en este caso, *incoming-call-dispatch*. Además, es necesario registrar aquellas funciones que se van a exportar al enclave utilizando el identificador de dicha función, y el nombre de la función wrapper.

Por último y antes de lanzar el enclave creado, es necesario inicializar el buffer de memoria compartida que se va a utilizar.

E.2.2. Eapp

Por otro lado, en la figura E.2 aparece el código de la *eapp*, disponible bajo la ruta */sdk/examples/nonce-generator/eapp/eapp_nonce.cpp*. En dicho código se inicializan cada uno de los campos del *struct* que se va a utilizar para enviar la información desde la *eapp* al exterior, al host *untrusted*. Aparece encuadrado en color rojo, la operación de movimiento de bits realizada para generar el *nonce*.

Después, se invoca al método *ocall-print-struct* que invoca al *runtime* para que ejecute la *ocall* correspondiente.

```
int main(){

        Cont con;
        memcpy(con.array0, "El", 3);
        memcpy(con.array1, "nonce", 6);
        memcpy(con.array2, "generado", 9);
        memcpy(con.array3, "por el", 7);
        memcpy(con.array4, "enclave", 8);
        memcpy(con.array5, "es: ", 5);

        unsigned bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1;
        unsigned short lfsr_new = (lfsr >> 1) | (bit << 15);

        lfsr = lfsr_new;

        con.number0 = lfsr;

        ocall_print_struct(&con);

        EAPP_RETURN(0);
}
```

Figura E.2: Código fuente *eapp*.

E.3. Edge Calls

Tal y como se ha comentado previamente, Keystone ofrece una API de comunicación para intercambiar información entre la zona *trusted*, desde la *eapp* al exterior, el host *untrusted* haciendo uso del buffer de memoria compartida.

En esta sección se va a profundizar sobre cómo se realiza dicha comunicación. Para ello, se recomienda seguir la figura E.3

Partiendo del código fuente de la *eapp* tal y como se aprecia en la figura E.2, aparece una invocación a la función *ocall_print_struct*. El código relativo a dicha función se puede observar en la figura E.4.

La función *ocall_print_struct* permite exportar una estructura de datos de tipo *struct* previamente definida en el listing 5.1, desde el enclave al host *untrusted*, para posteriormente ser mostrada por salida estándar por dicho host.

La *eapp* llama a la función *ocall_print_struct*, que es una función de tipo *edge wrapper* que acaba invocando al *runtime* mediante una instrucción *ecall* tal y como se puede observar en las figuras E.5 y E.6.

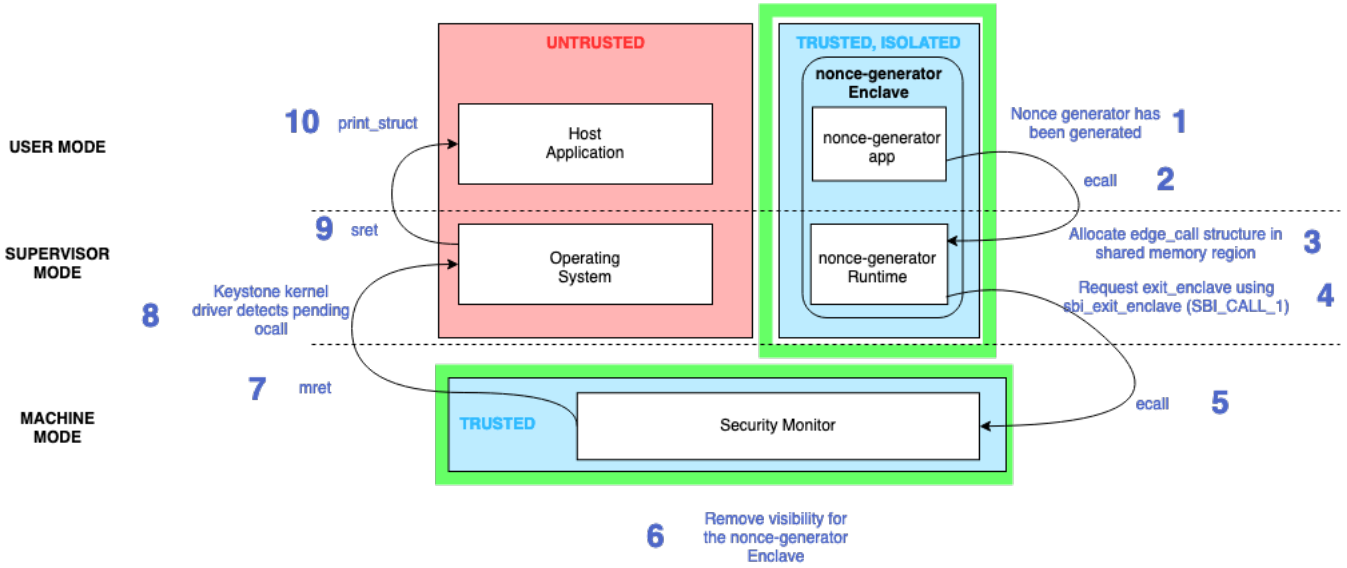

Figura E.3: Flujo de operaciones realizadas para pasar información y control desde *eapp* a host *untrusted*

```
unsigned long ocall_print_struct(Cont* stru){
        unsigned long retval;
        ocall(OCALL_PRINT_STRUCT, stru, sizeof(Cont), &retval, sizeof(unsigned long));
        return retval;
}
```

Figura E.4: Código fuente *ocall_print_struct*

Dicho *runtime* coloca una estructura de tipo *edge_call* en el buffer de memoria compartida rellenando el tipo de llamada, copiando los valores en otra zona del buffer y asignando el offset para los argumentos. Cabe destacar que las *edge calls* no utilizan punteros, sino que trabajan con offsets en el buffer de memoria compartida.

Finalmente, el *runtime* sale del enclave con una *SBI_CALL*, indicando que el enclave no está siendo destruido pero sí ha ejecutado una *ocall*.

En el lado del host *untrusted*, el kernel driver de Keystone, que trabaja con nivel de privilegio supervisor, comprueba el estado de salida del enclave, detecta una *ocall* pendiente, y pasa el control al *host untrusted* con una instrucción de tipo *sret*, que trabaja en nivel de privilegio usuario.

El *host untrusted* consume la estructura *edge_call* y la procesa con el método

```
int
ocall(unsigned long call_id, void* data, size_t data_len, void* return_buffer, size_t return_len) {
        return SYSCALL_5(SYSCALL_OCALL, call_id, data, data_len, return_buffer, return_len);
}

int
ocall_two(unsigned long call_id, void* data, size_t data_len, void* data1, size_t data_len1, void* return_buffer, size_t return_len) {
        return SYSCALL_7(SYSCALL_OCALL_TWO, call_id, data, data_len, data1, data_len1, return_buffer, return_len);
}
```

Figura E.5: Código fuente *ocall* en *syscall.c*. Aparece también el código relativo a la función *ocall_two* mencionada anteriormente.

Figura E.6: Código fuente *ocall* en *syscall.h*

encargado de gestionar dicha *ocall* (*print_string_wrapper*), registrado previamente tal y como se puede apreciar en la figura E.1 mediante el método *register_call*.

El wrapper genera un puntero al valor del argumento desde el offset en el buffer de memoria compartida y llama a *print_struct* con el valor pasado como argumento. Siguiendo la figura E.7, el método *print_struct* muestra la información por salida estándar.

```
unsigned long
print_struct(Cont* stru){
  return printf("Enclave said: %s %s %s %s %s %s %u \n",
                stru->array0,
                stru->array1,
                stru->array2,
                stru->array3,
                stru->array4,
                stru->array5,
                stru->number0);
}
```

Figura E.7: Código fuente *print_struct* en *host_nonce.h*

Para devolver el control al enclave, a la zona *trusted*, el wrapper setea el valor de retorno a éxito tras haber mostrado la información por salida estándar y devuelve el control al kernel driver, en modo supervisor. El kernel driver vuelve a entrar al enclave *runtime* mediante una *SBI_CALL*, previo paso de control al *SM* en modo máquina para hacer visible dicho enclave.

Una vez tiene el control de nuevo el *runtime*, reanuda la función *ocall_print_struct*, y devuelve el valor de retorno a la propia *eapp* en modo usuario, para lo cual habrá tenido que ejecutar una instrucción *sret*.