TESIS DE LA UNIVERSIDAD DE ZARAGOZA

2020 135

Jorge David de Hoz Diego

IoTsafe: Método y sistema de comunicaciones seguras para ecosistemas IoT

Departamento
Ingeniería Electrónica y Comunicaciones

Director/es
Ruíz Más, José de las Huertas
Saldaña Medina, José María

ISSN 2254-7606

Prensas de la Universidad
Universidad Zaragoza

ISSN 2254-7606

Universidad Zaragoza
1542

Tesis Doctoral

LOTSAFE: MÉTODO Y SISTEMA DE COMUNICACIONES SEGURAS PARA ECOSISTEMAS LOT

Autor

Jorge David de Hoz Diego

Director/es

Ruíz Más, José de las Huertas
Saldaña Medina, José María

UNIVERSIDAD DE ZARAGOZA

Ingeniería Electrónica y Comunicaciones

2020

Universidad Zaragoza
1542

Tesis Doctoral

IoTsafe: Método y sistema de comunicaciones seguras para ecosistemas IoT

Autor

Jorge David de Hoz Diego

Director/es

José Ruiz Mas
José Mª Saldaña Medina

Departamento de Ingeniería Electrónica y Comunicaciones
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Año 2020

Para alcanzar lo imposible,
uno debe intentar lo absurdo.

– **Miguel de Cervantes**

The way to be safe is never to be secure.
(La forma de estar a salvo es nunca sentirse seguro.)

– **William Shakespeare**

AGRADECIMIENTOS

Este trabajo no habría sido posible sin la dedicación y compromiso demostrado por José Saldaña, mi codirector de tesis y coautor de mis publicaciones. Puedo afirmar que ha contribuido decisivamene a mi formación y que todas sus aportaciones han resultado siempre fundamentales en los estudios llevados a cabo. También le agradezco a Jose Ruiz, mi director de tesis, la confianza depositada en mi propuesta desde el principio y el enfoque enriquecedor y pragmático que ha ofrecido a mis ideas. Me gustaría igualmente expresar mi agradecimiento a Julián, y al personal de la Unversidad que me han apoyado tanto desde la distancia con todo tipo de trámites.

De igual modo, quiero expresar mi agradecimiento a todos mis compañeros de Servicios de TI de Durango, y en particular a Joel, Emmanuel y José Ángel, que contribuyeron tan activamente en el desarrollo de los proyectos que inspiraron algunos de los aspectos técnicos defendidos en esta tesis. También quiero agradecer especialmente el apoyo que Roger me ha prestado en todo momento durante mis estancias en Chicago.

Quiero poner en valor la colaboración con el personal de la Universidad tecnológica de Durango en el proyecto de Conacyt que compartimos, y en particular reconocer especialmente el compromiso de Rebeca y Félix, que siempre estuvieron dispuestos a dedicar el tiempo que fuera necesario para el proyecto.

Así mismo, me gustaría agradecer a mis hermanos su apoyo incondicional, y a mi gatita Isis, por su obsesiva voluntad de querer ayudarme a terminar la tesis cada vez que dejaba el teclado del ordenador desatendido. Especialmente, estaré siempre agradecido a mi papá por el apoyo y confianza demostrados en todo momento, y por animarme desde el principio a patentar y poner en valor mis ideas.

Y a los que ya no están,

He de agradecer a mi abuela Manolita, por haber sido siempre un ejemplo de ingenio y superación ante el sin fin de adversidades que depara la vida.

Y a ti, mamá, por toda la confianza, ánimo, energía y cariño. Tu optimismo y valentía siempre me sirvieron de inspiración. Siempre orgulloso de ti y afortunado, porque siempre estuviste y siempre te recordaré.

ÍNDICE

ACRÓNIMOS

Acrónimo	Término
ABI	*Application Binary Interface*
API	*Application Programming Interface*
CoAP	*Constrained Application Protocol*
CoAPs	*Constrained Application Protocol secure*
Compatriot	*COordination and Management of Privacy And security for TRustable IoT*
Conacyt	Consejo Nacional de Ciencia y Tecnología de México
COTS	*Comercial Off-The-Shelf*
DDoS	*Distributed Denial of Service*
DI	*Deep Inspection*
DPI	*Deep Packet Inspection*
DS	*Digital Signage*
DTLS	*Datagram Transport Layer Security*
EAL7	*Evaluation Assurance Level 7*
ECDSA	*Elliptic-Curve Digital Signature Algorithm*
ELF	*Executable and Linkable Format*
ERP	*Enterprise Resource Planning*
FPGA	*Field-Programmable Gate Array*
GDPR	General Data Protection Regulation
GPIO	General-Purpose Input/Output
GPS	*Global Position System*
HDMI	*High-Definition Multimedia Interface*
HDMI-CEC	*Recommended Standard 232*
HMAC	*hash-based message authentication code*
HTTP	*Hyper Text Transfer Protocol*
HTTPS	*Hypert Text Transfer Protocol Secure*
I2C	*Inter Integrated Circuit*
IA	Inteligencia Artificial
IA	Inteligencia Artificial
ICE	*Interactive Connectivity Establishment*
ICMP	*Internet Control Management Protocol*
IEEE	*Institute of Electrical and Electronic Engineers*
IoT	*Internet of Things*
IP	*Internet Protocol*
ISA	Interconexión de Sistemas Abiertos
LSM	*Linux Security Module*
M2M	*Machine to Machine*
MITM	Man In The Middle

ML	*Machine Learning*
MMU	*Memory Management Unit*
MPLS	*Multi Protocol Label Switching*
MVC	Modelo Vista Controlador
NSA	*National Security Agency*
OSI	*Open Systems Interconnection model*
OTA	*Over The Air*
PHP	*Hypertext Preprocessor*
PKI	*Public Key Infrastructure*
POSIX	*Portable Operating System Interface*
PyMEs	Pequeñas y Medianas Empresas
RAM	*Random Access Memory*
REST	*Representational State Transfer*
RFID	*Radio Frequency IDentification*
ROM	*Read Only Memory*
RPS	*Requests per second*
RS232	*High Definition Media Interface-Consumer Electronic Control*
RTOS	*Real Time Operating System*
RTT	*Round Trip Time*
SDN	*Software Defined Network*
SSH	Secure Socket Shell
SSLB	*Second Stage Boot Loader*
SSO	Objetos de Seguridad de Sistema
STUN	*Session Traversal Utilities for NAT*
TCP	*Transmission Control Protocol*
TI	Tecnologías de la Información
TLS	*Transport Layer Security*
TOE	*Target of Evaluation*
UDP	*User Datagram Protocol*
USB	*Universal Serial Bus*
VPN	*Virtual Private Network*

GLOSARIO

Para facilitar la comprensión de IoTsafe, así como la de las implementaciones que sirven de ejemplos, se adjuntan a continuación las definiciones de los términos más relevantes que se usan en este documento. Toda referencia numérica que sea de la forma (1XX) es relativa a la Fig. 3.3-1 y toda referencia de tipo (4XX) es relativa a la Fig. C-11.

Acceso concedido a un *socket* protegido: Se denomina acceso concedido a un *socket* protegido, o simplemente "acceso concedido" a la configuración de un contexto de seguridad que habilite a un usuario de un sistema operativo para acceder a un *socket* protegido local. Un solo acceso concedido es aplicable a cualquier número de conexiones concurrentes realizadas desde la cuenta de ese usuario al *socket* protegido al que hace referencia dicho acceso.

Agente: Módulo *software* que permite al servidor *proxy-firewall* realizar diversas operaciones en un dispositivo remoto. Este *software* está configurado para:

- Iniciar las solicitudes de establecimiento de sesión de comunicación hacia el servidor *proxy-firewall* a través de un protocolo seguro.
- Realizar solicitudes de proxificación de *sockets* de forma directa y reversa.
- Establecer, mediante llamadas al sistema hacia el *framework* de red, la protección de los *socket* locales utilizados en las proxificaciones en el dispositivo_B.

El agente se denomina de distintas maneras según sea el dispositivo en el que se encuentre:

- Agente_A: (408) si se encuentra en un dispositivo_A. Sólo es necesario en el dispositivo_A si éste requiere realizar proxificaciones con otros dispositivos_A.
- Agente_AF: si se encuentra en un dispositivo_AF.
- Agente_B: (109) si se encuentra en un dispositivo dispositivo_B.

Alias: Identificador referido a un *socket* concreto, que sirve para nombrar el *socket* de un dispositivo. Es importante remarcar que el alias hace referencia al *socket* original que ofrece un servicio o recurso. Todas sus proxificaciones heredarán este alias, el cual puede ser de varios tipos:

- Alias_B: si el alias hace referencia a un *socket_B* de un proceso del dispositivo_B.
- Alias_A: si el alias hace referencia a un *socket_A* de un proceso del dispositivo_A.

Aplicación final: Aplicación utilizada por un usuario final.

Base de datos: (122) (414) Módulo *software* configurado para guardar y consultar distintos tipos de registros utilizados en el diseño, entre los cuales se encuentran: registro_*socket*, registro_acceso, registro_acceso_A, permiso_A_U y el permiso_A_G.

Cliente *proxy*: (112) (409) Módulo *software* complementario de cualquier agente que implementa el protocolo de comunicaciones seguro que se usa para el establecimiento de la sesión de comunicaciones y para realizar las proxificaciones, tanto directas como reversas.

Connection-broker: (121) (415) Módulo *software* empleado para mantener un registro y control sobre las proxificaciones realizadas y los accesos a las mismas en el dispositivo_A. También se ocupa de cerrar periódicamente los accesos inactivos.

Contexto de seguridad: Entorno de red local caracterizado por unos permisos de acceso a ciertos *socket* de red protegidos.

Contexto de seguridad amplio: Contexto de seguridad aplicado entre un *socket* protegido y el espacio usuario de múltiples usuarios del mismo sistema operativo pertenecientes a un mismo grupo. En IoTsafe, se genera un contexto de seguridad amplio cuando se aplica un permiso_A_G en un *socket* protegido.

Contexto de seguridad estricto: Contexto de seguridad aplicado entre un *socket* protegido y el espacio de usuario de una cuenta concreta del mismo sistema operativo. En IoTsafe, se genera un contexto de seguridad estricto cuando se aplica un permiso_A_U en un *socket-proxy* del dispositivo_A por medio de reglas del *framework* de red. También se generan en el dispositivo_B cuando se aplican las reglas descritas en el pefil_B_A.

Con_Id: Identificador del macado realizado en el tráfico de red relativo a la comunicación con un *socket* protegido. Este id determina la cuenta de usuario o el grupo de usuarios fuente de dicha comunicación, registrada en la base de datos del *connection-broker* con destino ese *socket* protegido específico.

Dispositivo_A: (106) (405) Elemento *hardware* que alberga al menos: un servidor *proxy-firewall*, un *connection-broker*, un *framework* de red, una base de datos, un perfil_A, y opcionalmente un agente_A. En IoTsafe, si algún proceso requiere de dos dispositivos_A, el dispositivo sobre el que se describe el procedimiento principal se constituye como el dispositivo_A y el otro como dispositivo_AF.

Dispositivo_AF: Elemento *hardware* equivalente en funcionalidades a un dispositivo_A. La razón de diferenciarlo como dispositivo_AF es facilitar la explicación de las comunicaciones que se puedan producir entre dos dispositivos_A.

Dispositivo_B: (101) Elemento *hardware* que alberga al menos un *framework* de red, un agente_B y un perfil_B_A. En caso de albergar también un servidor *proxy-firewall*, la configuración de operación de este módulo *software* se incluye también en el perfil_B_A, pues este servidor *proxy-firewall* tendrá funciones reducidas y orientadas a extender el control que el servidor *proxy-firewall* del dispositivo_A dispone de los módulos *software* del dispositivo_B, particularmente para procesos de actualización.

Entorno del usuario: Entorno del sistema operativo en el que aplican o tienen efecto las configuraciones y privilegios asociados al usuario de dicho entorno.

Espacio del *kernel*: (419) Área de memora donde operan exclusivamente los procesos del núcleo del sistema operativo.

Espacio del usuario: (418) Área de memoria donde operan los procesos de los usuarios del sistema operativo.

***Framework* de red**: (107) (420) Conjunto de herramientas que interactúan con el sistema operativo o que forman parte de él y que permiten configurar el tratamiento que reciben los paquetes de las comunicaciones que se cursan del espacio usuario al espacio del *kernel* y viceversa.

Host: Máquina física o virtual con sistema operativo multiusuario que alberga una o varias plataformas.

Id_Disp*Proxy*: Variable relativa al registro_*socket* de un *socket* protegido que identifica el id del dispositivo con el que se mantiene una sesión para poder acceder al *socket* referido por el alias y el id_Disp*Socket* del registro. En el caso particular de que el registro_*socket* se refiera a un *socket-proxy* de otro *socket-proxy*, el id_Disp*Socket* se referirá al id del dispositivo del segundo *socket-proxy*.

Id_DispSocket: Variable relativa al registro_*socket* de un *socket* protegido que identifica, junto con el alias del *socket* de dicho registro, el *socket* protegido del dispositivo. Esta variable hace referencia al id del dispositivo con el proceso que originó el *socket* que ofrece el recurso o servicio y puede almacenar tres tipos de identificadores:

- Un id_B, si el alias del registro_acceso refiere a un *socket_B*.
- Un id_AF, si el *socket* del registro_acceso comunica con un *socket-proxy*.
- Un id_A, si el alias del registro_acceso refiere a un *socket_A*.

Id_Usuario: Identificador de un usuario o de la cuenta relativa al mismo en el sistema operativo del dispositivo_A. Este identificador representa unívocamente al usuario y su cuenta de usuario, pudiéndose utilizar estos dos conceptos de forma indistinta en escenarios relativos a los contextos de seguridad del dispositivo_A.

Monitor de accesos: Módulo *software* empleado para mantener un registro y control sobre las proxificaciones realizadas y los accesos a las mismas en el dispositivo_A. Éste puede funcionar de dos maneras:

- Verificando periódicamente si existe tráfico relativo a un *socket* protegido, y en cuyo caso, actualiza el estatus de última_actividad del registro_acceso pertinente.
- Actualizando periódicamente el estatus de última_actividad del registro_acceso de los accesos que monitoriza para garantizar la accesibilidad a los *socket* protegidos respectivos mientras el proceso que solicitó los accesos siga en ejecución.

Cuando un monitor de accesos termina, bien al no haber actividad a lo largo de un periodo de tiempo determinado o porque el proceso responsable de las proxificaciones ha terminado, se dejan de actualizar los registros_acceso de los que se encargaba. Esto provoca que la inactividad de esos registros_acceso se incremente y eventualmente sean cerrados por el *connection-*

broker. Cuando esto ocurre, el servidor *proxy-firewall* es notificado para que borre las reglas de marcado pertinentes y cierre los accesos relativos a los registros_acceso borrados. Si los registros_acceso borrados se referían a una proxificación entre un dispositivo_A y un dispositivo_AF, el *connection-broker* también elimina el registro_*socket* y el permiso_A_U relativo a cada registro_acceso.

Perfil_A: Perfil de configuración del servidor *proxy-firewall* y del *connection-broker* del dispositivo_A.

Perfil_B_A: Perfil de configuración del agente del dispositivo_B para configurar la sesión de conexión al servidor *proxy-firewall* del dispositivo_A. En este perfil también se encuentran las credenciales para conectarse al dispositivo_A utilizando la cuenta de usuario usuario_A_B del sistema operativo del dispositivo_A. Así mismo, también incluye información relativa a la capacidad del usuario_A_B de solicitar proxificaciones en el dispositivo_A, siendo dicha capacidad validada por el servidor *proxy-firewall* tras cada solicitud de proxificación.

Perfil_UA_{AF}: Perfil de configuración para el agente del dispositivo_A para configurar la sesión de conexión al *proxy-firewall* del dispositivo_AF. Este perfil pertenece al usuario_A y en él también se encuentran las credenciales para conectarse al dispositivo_AF utilizando la cuenta de usuario usuario_A_{AF} del sistema operativo del dispositivo_AF. En este perfil también se encuentra información relativa a la capacidad del usuario_A_{AF} de solicitar proxificaciones en el dispositivo_AF, siendo dicha capacidad validada por el servidor *proxy-firewall* del dispositivo_AF tras cada solicitud de proxificación.

Permiso_A_G: Este tipo de permiso sirve para otorgar el acceso de un grupo de usuarios del dispositivo_A a un *socket_A* a través de un contexto de seguridad amplio, y comprende las siguientes variables:

- id_Grupo, que almacena el id del grupo de usuarios del sistema operativo del dispositivo_A al que se le otorga el permiso de acceso.
- alias, que almacena el alias del *socket_A*.
- *socket*, que guarda el valor del *socket_A*.

Permiso_A_U: Este tipo de permiso sirve para otorgar acceso al usuario_A a un *socket* protegido que sea un *socket-proxy* a través de un contexto de seguridad estricto. Se guarda como registro en la base de datos del *connection-broker* y dicho registro comprende las siguientes variables:

- id_Usuario, que guarda el id del usuario del sistema operativo del dispositivo_A.
- alias, que almacena el alias del *socket* protegido que se proxifica como *socket-proxy* en el dispositivo_A.
- id_DispSocket, que almacena el id del dispositivo con el *socket_B* o *socket_A* relativo al alias del *socket* que ofrece el recurso a través del *socket-proxy*.

Plataforma: (105) (413) Módulo *software* que permite monitorizar y administrar la información y servicios que son ofrecidos por otros dispositivos de la red. Esta información y servicios que ofrece la plataforma, principalmente se obtienen a partir de los dispositivos_B. La función principal de la plataforma consiste en procesar estos datos y servicios para ponerlos a disposición de los usuarios finales en forma de recursos propios. Por otro lado, también puede actuar de forma directa sobre los elementos de la red que gestione para modificar su configuración o requerir actuaciones de los mismos utilizando los *socket* protegidos locales a los que tenga acceso.

Proxificación directa: (114) Un tipo de proxificación que se produce cuando el *socket-proxy* resultado de la misma se genera en el dispositivo que inicia la sesión para ello, y ese *socket-proxy* ofrece recursos o servicios de un *socket* del dispositivo a donde se dirige el establecimiento de sesión.

Proxificación o *socket-proxy*: Resultado de la acción de proxificar, en el que se define un nuevo *socket* de red, por medio del cual es posible acceder al *socket* origen de la proxificación de forma transparente y obtener los recursos o servicios ofrecidos por el mismo.

Proxificación reversa: (115) Un tipo de proxificación que se produce cuando el *socket-proxy* resultado de la misma se genera en el dispositivo a donde se dirige el establecimiento de sesión para ello, y este *socket-proxy* ofrece recursos o servicios de un *socket* del dispositivo que inicia el establecimiento de sesión.

Proxificar: Acción de ofrecer el servicio o recurso provisto por un *socket* de red en otro *socket* de red diferente, pudiendo este *socket* ubicarse en un dispositivo remoto.

Proxy: (410) (423) Módulo *software* utilizado por cada tipo de protocolo empleado por las solicitudes de comunicación dirigidas a un *socket* protegido que no emplee el mismo tipo de protocolo de comunicación que el utilizado por el proceso origen de la solicitud. El *proxy* funciona como traductor entre protocolos para hacer la comunicación transparente y entre sus funciones figuran:

- Autenticar las solicitudes de comunicación con credenciales de algún usuario del sistema operativo del dispositivo_A que disponga de permiso_A_U o que pertenezca a un grupo con permiso_A_G que le permita acceder al *socket* protegido.
- Habilitar un registro_acceso en caso de ser necesario mediante llamadas al sistema.
- Retirar las credenciales de autenticación de la comunicación para que ésta pueda enviarse al *socket* protegido y dicho proceso de autenticación sea transparente a este último.

Registro_acceso: Un tipo de registro relativo a una cuenta de usuario del sistema operativo del dispositivo_A y que se genera por cada acceso concedido a un *socket* protegido de tipo *socket-proxy* directo o reverso, el cual comprende:

- Un con_Id, que guarda el id del acceso relativo a este registro.

- Una variable denominada "*socket*" que almacena el *socket* protegido al que se concede acceso y está formado por la dirección de red local y puerto.
- Un id_Usuario, que guarda el id del usuario al que se concede el acceso.
- Una última_actividad que inicialmente guarda el momento en el que se crea el registro y posteriormente se actualiza periódicamente (si procede) por el monitor de accesos.

Registro_acceso_A: Un tipo de registro relativo a un grupo de usuarios del sistema operativo del dispositivo_A, que se genera por cada acceso concedido a un *socket* protegido de tipo *socket_A*, y el cual comprende:

- Un con_Id, que guarda el id del acceso relativo a este registro.
- Una variable denominada "*socket*" que almacena el *socket_A*.
- Un id_Grupo, que guarda el id del grupo del usuario al que se concede el acceso.

Registro_*socket*: Un tipo de registro del dispositivo_A relativo a un *socket* protegido. Se guarda en la base de datos del *connection-broker* y consta de:

- Una variable denominada *socket*, que guarda el *socket* protegido al que hace referencia el presente registro_*socket*.
- Un alias del *socket* al que hace referencia el presente registro_*socket*.
- Un id_Disp*Socket* relativo al dispositivo con el *socket* al que hace referencia el alias.
- Un id_Disp*Proxy* que identifica el dispositivo al que conduce la sesión en la que se genera el *socket* protegido.
- Un tipo_*socket*.

Sandbox-jail: Conjunto de reglas de filtrado de paquetes que configuran una protección de acceso a los *sockets* protegidos. También restringe los destinos de las comunicaciones.

Servidor *proxy*: (113) (422) Módulo *software* complementario del servidor *proxy-firewall* que implementa el protocolo de comunicación seguro que se usa para el establecimiento de sesión y que recibe y procesa las solicitudes de proxificaciones, tanto directas como reversas.

Servidor *proxy-firewall*: (111) Un módulo *software* que está configurado para:

- Recibir las solicitudes de establecimiento de sesión de otros dispositivos a través de un protocolo seguro por medio de un servidor *proxy stand-alone*.
- Configurar contextos de seguridad estrictos y amplios mediante llamadas al sistema (427) al *framework* de red del sistema operativo del dispositivo_A.
- Procesar solicitudes de proxificación denominadas "*socket-proxy*" a través de sesiones establecidas o llamadas al sistema de procesos locales.
- Monitorizar los accesos a *socket* locales mediante monitores de acceso.
- Gestionar las actualizaciones de seguridad de los dispositivos con los que mantenga una conexión.

- Establecer, mediante llamadas al sistema hacia el *framework* de red, la protección de los *socket* locales utilizados en las proxificaciones en el dispositivo_A.

***Socket* protegido**: *Socket* de red local cuyo acceso se encuentra restringido en base a reglas de filtrado del *framework* de red del sistema operativo del dispositivo.

Socket_A: (123) *Socket* protegido del dispositivo_A que ofrece recursos o servicios a procesos locales y dispositivos remotos por medio de proxificaciones directas o reversas.

Socket_AF: *Socket* protegido del dispositivo_AF que ofrece recursos o servicios a procesos locales y dispositivos remotos por medio de proxificaciones directas o reversas.

Socket_B: (118) *Socket* protegido del dispositivo_B que ofrece recursos o servicios a dispositivos remotos a través de un *socket-proxy* reverso en el dispositivo_A.

Tipo_*socket*: Variable relativa al registro_*socket* de un *socket* protegido que identifica el tipo de *socket* al que hace referencia dicho registro pudiendo ser *socket-proxy* reverso, *socket-proxy* directo o *socket_A*. Los *socket-proxy* reversos y directos a su vez pueden ser también de tipo *socket_A*, *socket_B* y *socket-proxy*, indicando este último que el *socket* protegido es una proxificación de otro *socket-proxy*.

Última_actividad: Variable relativa a un registro_acceso de un usuario_A a un *socket-proxy* que define el último momento en el que hubo actividad relacionada con el *socket* protegido al que se hace referencia en el registro_acceso.

Usuario final: Persona física que requiere a través de una aplicación final un recurso, o solicita un servicio de una plataforma.

Usuario_A_B: Usuario del sistema operativo del dispositivo_A que es utilizado por el agente_B cuando éste se conecta al dispositivo_A utilizando las credenciales definidas en el perfil_B_A.

Usuario_A_{AF}: Usuario del sistema operativo del dispositivo_AF que es utilizado por el agente_A cuando éste se conecta al dispositivo_AF utilizando las credenciales definidas en el perfil_UA_{AF}.

FIGURAS

TABLAS

RESUMEN

La presente tesis se desarrolla en el sector de las telecomunicaciones y está relacionada particularmente con el campo de la Ingeniería Telemática. De forma más específica, el trabajo persigue diseñar y validar una tecnología de red mejorada de comunicación segura que pueda constituirse como alternativa a las existentes actualmente en determinados escenarios. El objetivo primario de esta red es desacoplar la seguridad de las aplicaciones de los dispositivos IoT a partir de la definición de contextos de seguridad, y facilitar las comunicaciones entre procesos remotos de forma cuasi-transparente. Estos contextos se configuran a partir de la separación de privilegios de entornos de tipo *Portable Operating System Interface* (POSIX), junto con marcado y filtrado de paquetes locales *Internet Protocol* (IP). Esta configuración se puede implementar fácilmente aprovechando el protocolo *Secure Socket Shell* (SSH) y tomando ciertas ideas de *Multi Protocol Label Switching* (MPLS), pero también de conmutación de circuitos. El campo inmediato de aplicación abarca las infraestructuras y ecosistemas IoT que cuenten con dispositivos POSIX con capacidad para soportar SSH, algo bastante factible en una mayoría de los ecosistemas del Internet de las Cosas ((*Internet of Things*, IoT)). Los principales beneficios de esta solución son acortar los tiempos de desarrollo, facilitar la conectividad y simplificar el mantenimiento y las actualizaciones de seguridad.

CONTRIBUCIONES CIENTÍFICAS

En esta sección se listan las contribuciones a revistas y congresos con revisión con pares, que se han derivado de la investigación de esta tesis. Se incluye también la referencia de la patente presentada.

Patente

J. De Hoz-Diego, "Secure Communication Method And System Using Network Socket Proxying". PCT Patent WO/2019/059754, 28 03 2019. [En línea]. Disponible en: https://patentscope.wipo.int/search/es/detail.jsf?docId=WO2019059754

Revistas indexadas en Journal Citation Reports

J. D. de Hoz-Diego, J. Saldana, J. Fernandez-Navajas and J. Ruiz-Mas, "IoTsafe, Decoupling Security From Applications for a Safer IoT," IEEE Access, vol. 7, pp. 29942 - 29962, 2019.

J. D. de Hoz Diego, J. Saldana, J. Fernández-Navajas and J. Ruiz-Mas, "Decoupling Security From Applications in CoAP-Based IoT Devices," IEEE Internet of Things Journal, vol. 7, no. 1, pp. 467 - 476, 2020.

Congresos internacionales

J. D. de Hoz Diego, J. Saldana, J. Fernández-Navajas, J. Ruiz-Más, R. Guerrero Rodriguez, F. d. J. Mar Luna and R. I. Guerrera González, "Leveraging on Digital Signage Networks to Bring Connectivity to IoT Devices," en TELCON UNI, Lima, 2015.

J. de Hoz-Diego, J. Saldana, J. Fernández-Navajas, J. Ruiz-Mas, R. Guerrero Rodriguez and F. J. Mar Luna, "SSH as an Alternative to TLS in IoT Environments using HTTP," en 2018 Global Internet of Things Summit (GIoTS), Bilbao, 2018.

1. INTRODUCCIÓN

1.1. MOTIVACIÓN

En los últimos años, la irrupción de las comunicaciones entre dispositivos M2M (*Machine to Machine*) conduce a predicciones de alrededor de cientos de millones de dispositivos conectados conformando el Internet de las Cosas [1]. El constante crecimiento del número de dispositivos y automatismos derivados del auge del IoT [2], junto con el despegue de la Industria 4.0 [3] ponen de manifiesto la importancia de simplificar el control de las comunicaciones y la gestión de su seguridad. Antes de la aparición de estos paradigmas, el número de dispositivos desatendidos en cualquier tipo de proyecto era sensiblemente inferior, lo que presuponía una interacción directa entre los dispositivos con el usuario final o el operador de red en mayor o menor grado. Sin embargo, el elevado número de elementos que conforman estas redes y su previsión de crecimiento exponencial [1], sugiere que se pueda llegar a la cifra de los 500 millones de dispositivos conectados en 2030 [4]. Por ese motivo, se requiere que determinados aspectos de seguridad funcionen de forma sólida, estandarizada y confiable [5] para evitar que, por ejemplo, un supervisor de red tenga que analizar continuamente, elemento a elemento, el estatus de cada dispositivo. En todo caso, todas las iniciativas encaminadas a la mejora de la seguridad, deben evitar perjudicar la escalabilidad e interconectividad, para potenciar todas las funcionalidades potenciales de la IoT.

Existe una falta de previsión por parte de las empresas fabricantes de tecnología IoT en los costes a medio y largo plazo, requeridos para lograr una adecuada gestión de la seguridad de los dispositivos IoT. Este hecho conduce a que muchos consumidores opten por seguir trabajando con dispositivos obsoletos en aspectos relativos a su seguridad [6]. En muchas ocasiones, estos dispositivos no gestionan información sensible (por ejemplo, ciertas redes de sensores) por lo que, aún en el caso de reconocerse esos dispositivos como vulnerables, el usuario final opta por mantenerlos, ya que no suponen ningún riesgo real para él. Todo esto conduce a un escenario en el que infraestructuras críticas ajenas acaban corriendo el riesgo de ser atacadas [7] por *botnets*[1] [8], creadas precisamente a raíz de estos dispositivos IoT vulnerables [9] que han logrado ser infectados con código malicioso, pero que aparentemente continúan funcionando con normalidad.

Como consecuencia, los distintos gobiernos y cuerpos reguladores se ven obligados a legislar al respecto, para evitar que las carencias de seguridad en dispositivos IoT comerciales acaben suponiendo una amenaza a escala global [10]. La Comisión Europea, junto con La Agencia Europea de Seguridad de las Redes y de la Información, está realizando notables esfuerzos por abordar el problema de la seguridad a medio y largo plazo en estas tecnologías

[1] Botnet: conjunto de dispositivos informáticos conectados a Internet que ejecutan un software de manera automática y coordinada con un fin.

emergentes, tal como corroboran consultores externos [11]. En el corto plazo es muy probable que surja una legislación y regulación [12] [13] de alcance equiparable a la Regulación General de Protección de Datos o *General Data Protection Regulation* (GDPR) [14]. Esta nueva regulación trataría de imponer la obligatoriedad de certificaciones de seguridad de los dispositivos, con el fin de minimizar los riesgos de seguridad a lo largo de su vida útil real. Para ello, tanto la Comisión como sus agencias trabajan conjuntamente en un *framework* de Ciberseguridad [6] que permita afianzar un mercado único digital con garantías.

En la sección 1.2 de este capítulo se analizan brevemente los principales aspectos que determinan la problemática de seguridad actual en el campo de la IoT, y en la sección 1.3 se desarrollan los objetivos de la tesis como respuesta a dicha problemática.

1.2. PROBLEMÁTICA EN EL CONTEXTO ACTUAL

En este entorno hiperconectado, las principales características y prestaciones perseguidas al desarrollar nuevos productos han sido principalmente el coste unitario de los dispositivos IoT y un bajo consumo energético que favorezca una independencia, versatilidad y movilidad por mayor tiempo [15] [16] [17]. Sin embargo, estas prestaciones están empezado a quedar relegadas a un segundo plano frente a otras como la conectividad y sobre todo la seguridad. Se puede concluir que el desarrollo futuro de la tecnología IoT puede verse frenado notablemente si la seguridad no se toma seriamente en cuenta. Ciertamente, la carencia de seguridad puede incurrir en costes ocultos que las empresas tecnológicas desarrolladoras y los usuarios potenciales tienen que abordar a medio plazo.

La seguridad en las comunicaciones en ecosistemas IoT, particularmente en dispositivos con capacidad de ser actualizados, se plantea actualmente como una prestación que debe proporcionar el *software*/*firmware* de cada dispositivo. Dicha prestación, debe desarrollarse en conjunción con la plataforma *software* que se encargue de gestionar y administrar los dispositivos IoT, y de la que se deriven recursos y servicios relacionados. Independientemente del protocolo de comunicaciones empleado, la mayoría de las soluciones actuales se basan en embeber el *software* de autenticación y cifrado de las comunicaciones en el propio *firmware* del dispositivo o en la aplicación principal que éste utiliza [18]. Este *software* proporciona una protección criptográfica del canal de comunicaciones para evitar ataques de seguridad en algún punto intermedio evitando, entre otros, los siguientes problemas [19]:

- Que la información que se intercambia pueda ser escuchada (*eavesdropping, Fig. 1.2-1*).

Fig. 1.2-1: Eavesdropping: El atacante debe poder entender la comunicación que resulta intervenida.

- Que el receptor con el que se comunica el emisor de la comunicación suplante la identidad del auténtico receptor de la misma, y este agente usurpador, a su vez, se haga pasar por el emisor de la comunicación de cara al receptor (*man-in-the-middle, MITM Fig. 1.2-2*).

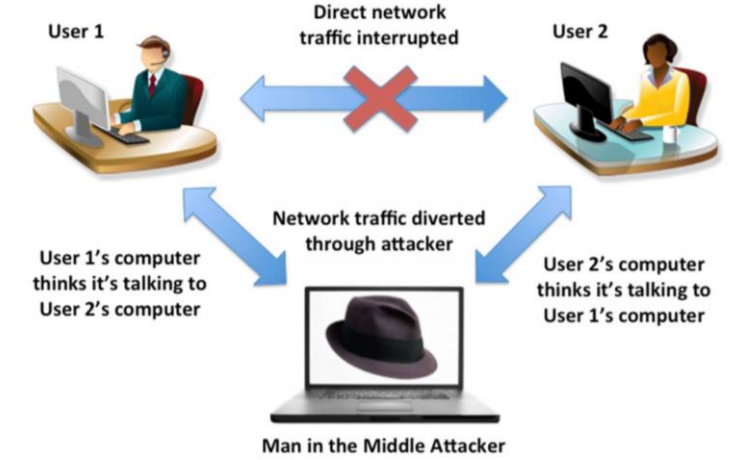


Fig. 1.2-2: Man-In-The-Middle: El atacante hace de intermediario entre los elementos que se comunican sin que ninguna de las partes pueda dar cuenta de ello[2].

[2] Ida Matero, "Ethical Hacking: Research and Course Compilation", 2016, https://www.theseus.fi/bitstream/handle/10024/119594/Matero_Ida.pdf

- Que la información de una comunicación, que normalmente debería fluir íntegramente entre emisor y receptor (Fig. 1.2-3), pueda modificarse, replicarse, alterarse su orden o falsificarse sin que este hecho sea detectado por el receptor de la comunicación (*tampering, Fig. 1.2-4*).

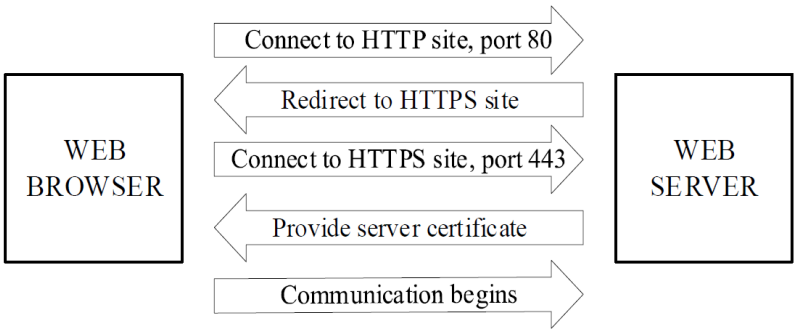


Fig. 1.2-3: Ejemplo de comunicación HTTP->HTTPS normal [20]***.***

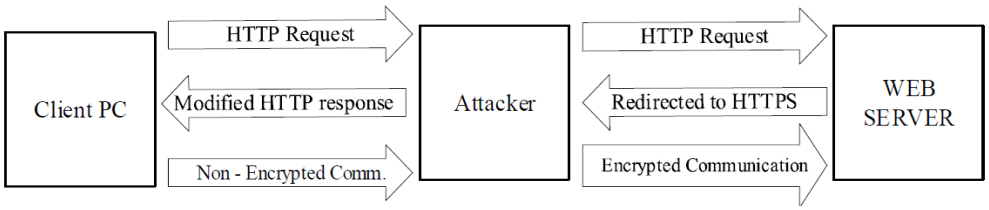


Fig. 1.2-4: Ejemplo de comunicación HTTP->HTTPS en la que se efectúa un ataque MITM para posteriormente y de forma inadvertida poder escuchar las comunicaciones (eavesdropping) y alterar la comunicación (tampering) [20]***.***

La protección criptográfica necesaria en la mayoría de las ocasiones se consigue utilizando un entorno *(framework)* de seguridad en el desarrollo de las aplicaciones que utilice *Transport Layer Security* (TLS, Fig. 1.2-5) [21] o algún protocolo de seguridad propietario [22]. *Datagram Transport Layer Security* (DTLS) [23] ofrece servicios equivalentes a TLS, pero para aplicaciones que trabajan con UDP. En cualquier caso, una modificación requerida para solucionar una vulnerabilidad que necesite actualizar algoritmos [24] o determinadas configuraciones de los aspectos de seguridad, suele requerir modificar y actualizar la aplicación completa y, en muchos casos, también el *firmware* del dispositivo. Algunos ejemplos de este tipo de implementaciones se encuentran en dispositivos que utilizan protocolos con seguridad como *COnstrained Application Protocol* (COAP), *Message Queuing Telemetry Transport* (MQTT), *Hypertext Transfer Protocol* (HTTP) o su versión 2 (HTTP/2).

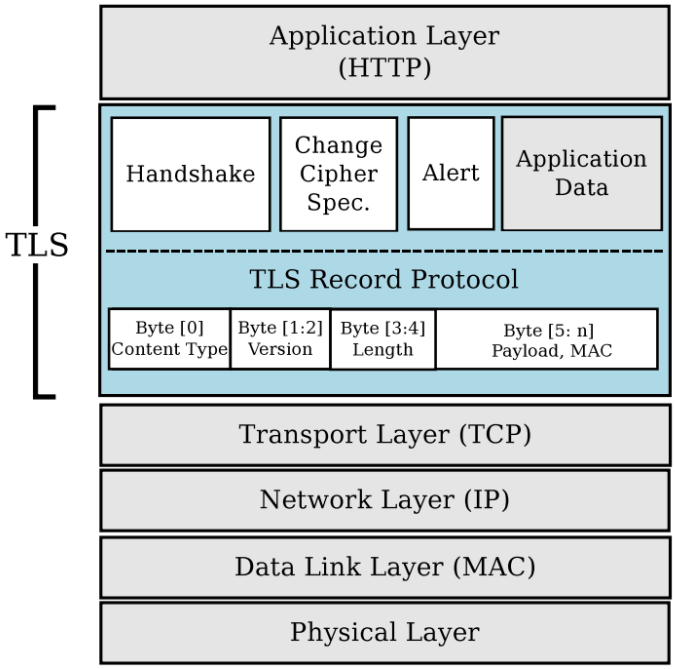


Fig. 1.2-5: Estructura de TLS como protocolo de seguridad dentro de las capas del modelo "Open Systems Interconnection model" (OSI) y sus sub-protocolos [25]**.**

Este tipo de diseño, en el que el protocolo de seguridad se encuentra integrado en el *software*, conduce a la necesidad de mantener activos los equipos de desarrollo de ese *software* para todas sus versiones y para cada modelo de dispositivo. Esta realidad justifica que las empresas fabricantes tiendan a limitar en el tiempo el soporte que dan a los productos que comercializan, animando a sustituir el dispositivo completo por su nueva versión, en caso de quedarse obsoleto su *firmware* o la seguridad de la aplicación. En el medio plazo, esta idea no suele representar una alternativa real para el consumidor final en muchos casos, particularmente en el mundo empresarial. Esto se debe a que no se puede contemplar únicamente el coste de los nuevos modelos de dispositivos, sino que pueden resultar mucho más importantes los costes derivados del cambio de los dispositivos antiguos por los nuevos: sustitución física de los dispositivos, coste de la interrupción de servicios, nueva formación para los empleados, actualización de los procesos internos, *software* de gestión/control, *middleware* dependiente, etc.

Paralelamente, al mismo tiempo que la cantidad de dispositivos IoT desplegados aumenta exponencialmente, estos elementos pueden convertirse en herramientas de los *hackers* para realizar potentes ataques de denegación de servicio (*Distributed Denial of Service*, DDoS): estos dispositivos constituyen redes de elementos heterogéneos que comparten vulnerabilidades y que, al encontrarse geográficamente distribuidos, facilitan el poder realizar ataques masivos a infraestructuras críticas de negocios y gobiernos [7]. Las vulnerabilidades potenciales que pueden ser aprovechadas por los *hackers*, fuerzan a que los usuarios de estos dispositivos IoT tengan que parchearlos con las actualizaciones de seguridad correspondientes. Si estas no existieran, idealmente tendrían que desconectarlos de Internet o reemplazarlos por otros nuevos.

Tras las filtraciones realizadas por Edward Snowden en 2013, se pusieron de manifiesto numerosos programas de espionaje gubernamental [26] como los de la NSA (*National Security Agency*), a través de los cuales era posible obtener información confidencial (pese a encontrarse cifrada) de distintos proveedores de servicios de Internet a través de programas de espionaje como Muscular [27]. Este tipo de espionaje se basaba en explotar vulnerabilidades de los protocolos de seguridad y en el uso de instalaciones y personal previamente comprometido por la NSA. Específicamente, se aprovechaba el fallo de no cifrar el tráfico en redes privadas de estas empresas [28]. Tras hacerse públicos estos informes, las principales empresas del sector tecnológico y de las comunicaciones objeto de estos ataques, pusieron en marcha de manera inmediata medidas de cifrado interno para mitigarlos.

Sin embargo, la vulnerabilidad en los servidores y las aplicaciones va más allá de redes inseguras o de una mala implementación de las mismas. El propio desarrollo de éstas puede verse también comprometido[3]. Las principales causas de las vulnerabilidades que se pueden encontrar en los dispositivos hoy en día se derivan de la falta generalizada de concienciación de la importancia de la seguridad, tanto de los desarrolladores como de los usuarios [29]. Desarrollos inseguros de dispositivos IoT que se entregan con combinaciones de *login* y *password* por defecto, pueden convertirse en objetivos de ataques automatizados de diccionario o de fuerza bruta realizados por *bots*. Estos dispositivos, una vez hackeados, pueden incorporase a una red de *bots* preparada para ejecutar ataques coordinados de denegación de servicio, o también pueden acabar infectados con programas *malware* más sofisticados, con otro tipo de objetivos.

Estos tipos de vulnerabilidades son fruto de tendencias generalizadas en el desarrollo de *software* y de dispositivos IoT escasamente comprometidas con la seguridad y contra las que resulta complicado luchar. Principalmente, se tiende a primar la simplificación del desarrollo y reducción de sus tiempos a base de:

- El uso de *software* y *hardware* genérico.
- Reducción de las pruebas de calidad y su alcance.
- Limitar el soporte técnico en el tiempo.

Luchar contra las motivaciones de este tipo de estrategias resulta una tarea compleja y prolongada en el tiempo, ya que las acciones a tomar pueden chocar con indicadores de rentabilidad a la que las empresas no pueden renunciar fácilmente de manera unilateral.

La complejidad derivada de mantener un dispositivo actualizado y seguro requiere importantes esfuerzos, incluso entre empresas especializadas del sector [30]. Por otro lado, los desarrolladores se ven empujados constantemente a optimizar sus tiempos de programación basándose en lenguajes de más alto nivel, *framework* diversos y repositorios de paquetes y de

[3] "Chips under the microscope". *Nat Electron* 2**,** 429 (2019), https://www.nature.com/articles/s41928-019-0325-z

librerías de desarrollo. Esto permite por ejemplo que antiguas técnicas de *hacking* como el "*typosquatting*"[4] estén cobrando nueva relevancia, al permitir sortear la cadena de seguridad y confianza entre proveedores de *software* [31] y, en particular, infectando repositorios de paquetes de lenguajes de programación como Python [32].

Por otro lado, el tiempo que transcurre desde la publicación de una vulnerabilidad hasta la implementación de las contramedidas pertinentes en las aplicaciones resulta muy variable, pudiendo llegar a tomar incluso años [33]. No obstante, el tiempo total necesario depende en gran medida de la aplicación en cuestión y del grado de actividad del equipo de desarrollo que ofrece el soporte. Por ejemplo, la mayoría de navegadores web solucionan dichas vulnerabilidades en un plazo mucho menor (semanas-meses) [34].

Finalmente, cabe destacar que todas las medidas de seguridad que se tratan de implementar de forma unilateral por fabricantes específicos (protocolos propietarios, nuevos algoritmos de seguridad, etc.) son generalmente muy costosas y con un alcance limitado a su gama de productos. Esta estrategia ofrece un nivel de seguridad difícil de certificar y de auditar por las entidades gubernamentales pertinentes, lo que complica la labor de los cuerpos reguladores de los diferentes países a la hora de elaborar normativas efectivas. Además, este tipo de medidas de seguridad acaba dificultando también la interconexión entre diferentes despliegues IoT, al requerir de complejos *middleware* intermedios que además suelen resultar costosos de mantener. Esto se debe principalmente a las cracterísticas específicas no comunes de dichos *middleware*, que encarecen sus actualizaciones, unido a que los requisitos de seguridad evolucionan con el tiempo cada vez más rápidamente. Además, las aplicaciones relacionadas no suelen ser independientes unas de otras por culpa de los algoritmos y protocolos para una comunicación segura, por lo que una actualización en un eslabón de la cadena acaba requiriendo una actualización más generalizada.

Antes de iniciar esta investigación, ya era evidente el compromiso de Europa para hacer frente a los retos de Internet del Futuro[5]. De hecho, este compromiso se ha materializado en importantes inversiones en proyectos de investigación, muchos de los cuales siguen activos, contando con expertos en diferentes disciplinas científicas. Además, se prevé que estas inversiones sigan produciéndose[6]. El IoT jugará un papel muy relevante en este campo debido a las características intrínsecas de los ecosistemas que lo forman, pues estos surgen de integrar tecnologías muy heterogéneas. El IoT se puede definir como una red mundial de objetos interconectados que incluye, entre otros, dispositivos de consumo como ordenadores, teléfonos móviles, y otras tecnologías como *Radio Frequency IDentification* (RFID), redes de sensores inalámbricos, actuadores, dispositivos inteligentes, Cyber Physical Systems (CPS [35]) la

[4] Typesquashing: Incluir en repositorios de confianza paquetes con nombres parecidos a los originales, pero con malware. De este modo, si el programador se equivoca al escribir el nombre de la dependencia o de la librería, el programa resultante compila y funciona con normalidad, pero incorporando también el malware inadvertidamente.
[5] "Future Internet", H2020 Horizon, https://ec.europa.eu/programmes/horizon2020/en/h2020-section/future-internet
[6] "European IoT Large-Scale Pilots", https://european-iot-pilots.eu/

Industrial Internet of Things (IIoT [36]), etc. Estas redes deberán ser capaces de trabajar dinámicamente para ofrecer servicios derivados de las sinergias que surgen de su colaboración.

Se puede concluir afirmando que la tecnología IoT se encuentra actualmente en una encrucijada. Dispone de un potencial de crecimiento exponencial reconocido para los próximos años, pero al mismo tiempo se ve lastrada por diversos problemas coyunturales de una tecnología emergente. A continuación, se profundiza en dichos problemas.

1.2.1. **Causas de la falta de seguridad**

Las autoridades y empresas tecnológicas experimentan actualmente una cierta pérdida de credibilidad ante la imposibilidad de detener y mitigar los efectos de la creciente escalada de algunos tipos de ataques [37] y de perseguir a los responsables con eficacia[7]. Algunos ejemplos de estos problemas consisten en filtraciones y pérdidas de datos personales y de recursos por vulnerabilidades explotadas por atacantes, que ocurren muchas veces por un tratamiento inadecuado de los datos confidenciales por parte de empresas e instituciones: datos privados de cuentas de Uber[8], datos de tarjetas de crédito[9], *ramsonware* en NHS[10], problemas con el *software* de gestión de los reactores nucleares de KHNP [38], fraudes con métodos tradicionales de pago [39], ataques de denegación de servicio distribuida, etc. Pese a que todavía no figura la IoT como causa raíz de dichos incidentes, algunos expertos coinciden en advertir que la tendencia empeorará en el futuro inmediato por la hiper-conectividad que traerá 5G [40].

La existencia de vulnerabilidades en la seguridad de las comunicaciones es un tema recurrente que se ha estudiado históricamente [41] de lo que se puede concluir lo siguiente:

- Aquellos elementos *software* más usados, son de los que se acaba descubriendo un mayor número de vulnerabilidades. Por consiguiente, una tecnología madura es estadísticamente más resistente a ataques que nuevos sistemas emergentes.
- Cuanto más simple sea el *software* desarrollado y el sistema operativo en el que funcione, menos vulnerabilidades pueden aparecer.
- Siempre se debe considerar que un *software* presenta vulnerabilidades ocultas y se debe contemplar de antemano una estrategia con la que abordar esas posibles vulnerabilidades en función del riesgo potencial de seguridad aparejado (*K-zero day* [42]).

El uso de la innovación para competir [43] fomenta el desarrollo de nuevos servicios y tecnología en los ecosistemas IoT, lo que a su vez fomenta que la tipología de las amenazas se

[7] "FBI Public Service Announcement, I-052518-PSA", mayo de 25, 2018, https://www.ic3.gov/media/2018/180525.aspx
[8] Robert N. Charette, "FTC Puts Uber on a Short Leash for Security Breaches", *IEEE Spectrum*, 20 de abril de 2018, https://spectrum.ieee.org/riskfactor/computing/software/ftc-puts-uber-on-a-privacy-and-security-choke-chain
[9] "Information on the Capital One Cyber Incident", 2019 https://www.capitalone.com/facts2019/
[10] "IEEE Global Internet Policy Monitor", 1 de noviembre de 2017, https://internetinitiative.ieee.org/resources/ieee-global-internet-governance-monitor?start=45

encuentre en constante evolución [44]. Los tiempos de programación y desarrollo son cada vez más cortos, lo que conduce a una híper-dependencia de los *framework* de desarrollo, *3rd party libraries*, repositorios de paquetes externos, etc. La innovación centrada tradicionalmente en las funcionalidades principales del dispositivo o servicio contribuye a que el rol de la seguridad se difumine entre los distintos elementos que constituyen el *software* utilizado en el desarrollo de las aplicaciones de los dispositivos IoT. La reutilización de paquetes de software, uso de *3rd party libraries*, SDKs de desarrollo y *frameworks* de programación específicos buscan acelerar y simplificar el diseño y elaboración de software. Esto causa que la responsabilidad que cada uno de estos elementos juega en la seguridad no quede muy definido, complicando la respuesta frente a vulnerabilidades que se vayan haciendo públicas.

Este efecto, sin embargo, no parece resultar muy relevante en el proceso de desarrollo del software. Lo primordial suele ser la inmediatez: el competidor que logra llegar con su producto o servicio al mercado antes que su competencia, gana hasta un 30% más de cuota de mercado que el resto que llegue después[11,12]. El resultado de esta dinámica se traduce en un ciclo de vida útil muy corto de algunos dispositivos (de entre 2-5 años) pese a que el *hardware* usado puede operar por muchos más (*The Internet of Trash*[13]). Este hecho se explica también en parte por las notables dificultades existentes para aplicar las actualizaciones de los dispositivos de forma efectiva[14] ya que muchas veces los dispositivos se comercializan "tal cual" ("*as is*"[15]).

Por otro lado, la tendencia extendida de utilizar *hardware* de propósito general de tipo *Comercial Off-The-Shelf* (COTS) junto con sistemas operativos de tipo Linux o *Real Time Operating Systems* (RTOS) [15], puede conducir a que los desarrolladores incluyan inadvertidamente brechas de seguridad en los dispositivos, si esta integración no se realiza adecuadamente [45]. Pese a la aparente mayor especialización requerida por los desarrolladores debido al constante avance tecnológico, dicha especialización se fundamenta en lenguajes de alto nivel en los que operan los *frameworks* de abstracción emergentes nacidos para acelerar dicho desarrollo, sin contar en muchas ocasiones con controles de calidad adecuados [46]. Por ello, los desarrollos de *software* IoT incluyen cada vez con más frecuencia errores ocultos que conllevan a asumir de forma general una política de desarrollo incremental basada en "parches",

[11] M. Ringel , A. Taylor , and H. Zablit, "The Rising Need for Innovation Speed", *Boston Consulting Group*, 2015, https://www.bcg.com/publications/2015/growth-lean-manufacturing-rising-need-for-innovation-speed.aspx

[12] Cisco, "The Hidden Costs of Delivering IIoT Services", *CISCO Jasper White Paper*, 2017, https://www.cisco.com/c/dam/m/en_ca/never-better/manufacture/pdfs/hidden-costs-of-delivering-iiot-services-white-paper.pdf

[13] Stacey Higginbotham, "The Internet of Trash: IoT Has a Looming E-Waste Problem", *IEEE Spectrum*, 2018, https://spectrum.ieee.org/telecom/internet/the-internet-of-trash-iot-has-a-looming-ewaste-problem

[14] FDA: U.S. Food and Drugs administration, "Firmware Update to Address Cybersecurity Vulnerabilities Identified in Abbott's (formerly St. Jude Medical's) Implantable Cardiac Pacemakers" *FDA Safety Communication*, 2017 https://www.fda.gov/medical-devices/safety-communications/firmware-update-address-cybersecurity-vulnerabilities-identified-abbotts-formerly-st-jude-medicals

[15] "A*s is*": Terminología utilizada para indicar que no existen garantías que se extiendan más allá de la descripción que figure del producto descartando cualquier otra garantía implícita (como por ejemplo, el soporte a medio/largo plazo) https://www.law.cornell.edu/ucc/2/2-316

pese a que un abuso de esta metodología puede llevar a multitud de errores críticos y desprestigio de la marca[16].

Esta forma de desarrollo incremental tan despreocupada incentiva la aparición de *bots* con los que se pueden automatizar rutinas de *hacking* que exploten vulnerabilidades de reciente publicación y que sean de amplio alcance. El éxito de esta estrategia de ataques se debe a la existencia en Internet de numerosos grupos de dispositivos IoT con una arquitectura *hardware* y sistema operativo común. Esto acaba favoreciendo la aparición de vulnerabilidades comunes; las cuales son el objetivo prioritario de esos *bots* para lograr *hackear* esos dispositivos e incluirlos en *botnets* [47] [48] (ej. Redes *Mirai* [49]). Estas *botnets*, a su vez se utilizan periódicamente para lanzar, entre otros, ataques de denegación de servicio principalmente contra infraestructuras críticas, frente a los cuales no resulta trivial el defenderse por su alto número, elevada dispersión geográfica de las fuentes.

Pese a todo, los usuarios, empresas y administraciones en general, están abrazando los dispositivos, tecnología y servicios que ofrece el mundo IoT, aunque con mayor lentintud de lo inicialmente previsto. Esto se debe a que se continúa recelando del uso de la tecnología emergente para actividades de especial relevancia. Se considera que, si se acepta el uso de las tecnologías IoT con un fin determinado, se está asumiendo implícitamente el riesgo que se corre por usar esa tecnología y la posibilidad de convertirse en una posible víctima[17] [50]. Por el momento, este hecho autorregula su uso en el corto plazo limitándolo a funciones percibidas como más triviales. Desafortunadamente, cuando los dispositivos no manejan información sensible del usuario, el control que éste realiza del mismo disminuye notablemente, lo que se traduce en un menor interés por actualizarlo o sustituirlo cuando es necesario por motivos de seguridad [9].

Por desgracia, en los ecosistemas IoT se dan algunos factores que además favorecen el grado de inseguridad de los sistemas involucrados. En primer lugar, los protocolos de seguridad empleados son también estándar y muy utilizados (TLS/DTLS). Pese a su amplia variedad de configuraciones y algoritmos, la configuración predeterminada de fábrica en TLS y DTLS es la que suele estar presente durante toda la vida útil del dispositivo, ya que las características de estos protocolos no suelen ser configurables una vez ya quedan implementados en la aplicación. Al considerarse ésta una solución muy utilizada, las vulnerabilidades que se descubren, tienden a corregirse con relativa rapidez. Sin embargo, dichas correcciones se terminan aplicando en el *software*/*firmware* de cada dispositivo en tiempos distintos dependiendo en gran medida de la motivación de la empresa desarrolladora del producto y en última instancia del usuario final.

[16] Mark Gurman, “Inside Apple’s iPhone Software Shakeup After Buggy iOS 13 Debut”, *Bloomberg Technology*, 2019, https://www.bloomberg.com/news/articles/2019-11-21/apple-ios-14-features-changes-testing-after-ios-13-bugs

[17] Consumers Internatonal and The Internet Society, “The trust opportunity: Exploring Consumers’ Attitudes to the Internet of Things”, 2019, https://www.internetsociety.org/wp-content/uploads/2019/05/CI_IS_Joint_Report-EN.pdf

1.2.2. **Regulaciones emergentes**

La rápida expansión de la tecnología y su incorporación a los sistemas industriales y de consumo han favorecido un espacio de desregulación en el ámbito de la seguridad. Esta tendencia fomenta la proliferación de "soluciones tecnológicas verticales" (*verticals*) que constituyen la estrategia más inmediata y utilizada para abordar un problema concreto con la ayuda de tecnología IoT. Sin embargo, este tipo de alternativas acaba minando la interoperabilidad entre los propios *verticals* por la complejidad inherente al *middleware* necesario para proporcionar la seguridad requerida para su comunicación. La utilización de protocolos propietarios y soluciones no suficientemente probadas en entornos productivos constituyen, por otro lado, la parte responsable de las cada vez más frecuentes situaciones de inseguridad y ataques masivos contra las que resulta tan difícil actuar [51] [52]. Este resultado, además de complicar la gestión de la seguridad perjudica la colaboración entre mercados, resultando profundamente perjudicadas las sinergias y nuevos modelos de negocio que pudieran surgir de dichas colaboraciones [53].

La inversión en seguridad necesaria cobra especial relevancia si se entiende como una medida preventiva que evite problemas hipotéticos en un futuro. Pese a que la concienciación actual existente resulta mayor que en años anteriores, en el caso particular de las PYMEs por ejemplo, resulta complicado que inviertan en la seguridad que realmente necesitan y que los clientes finales perciban también la importancia que estas medidas conllevan [54]. Un caso particular que conviene mencionar es el coste de reemplazar los dispositivos IoT obsoletos a nivel de seguridad, por otros más adecuados cuando la actualización no es viable. Las cifras tienden a dispararse por su gran número, dispersión geográfica, necesidad de equipos especiales complementarios para llevar a cabo la tarea, mano de obra cualificada o el propio coste derivado de la interrupción del servicio prestado durante el proceso [55].

Cuando sí es factible dicha actualización, la estrategia en seguridad seguida por la mayoría de empresas desarrolladoras de soluciones IoT suele ser de tipo "reactiva" a base de "*parches"*. Sin embargo, a diferencia de otros productos de TI, muchos de estos elementos trabajan 24/7 sin supervisión personalizada; lo que implica que a una gran parte de ellos se les tarde en aplicar dichas actualizaciones. Por eso mismo, esta estrategia reactiva puede considerarse insuficiente [54], necesitando complementarse con metodologías basadas en la seguridad del propio diseño "*secure-by-design*" [56]. Por otro lado, tampoco existen incentivos reales del proveedor o fabricante para extender la vida útil de un dispositivo, lo que puede resultar poco atractivo porque requiere la dedicación de recursos humanos de la empresa para ofrecer un mantenimiento específico por cada versión de aplicación y dispositivo [57]. En medios electrónicos como IEEE Spectrum, es común encontrar recientemente artículos de opinión alertando de la problemática de la seguridad, documentados con ejemplos y casos diversos [5]. "*Dado que los impactos de estos ataques no afectan a los fabricantes de IoT, se puede considerar que los problemas de*

seguridad no se resolverán confiando únicamente en las reglas de mercado. Se requiere que los gobiernos del mundo contribuyan para poder garantizar unos dispositivos IoT seguros por medio de incentivos a las compañías e inversión en investigación y desarrollo conjuntamente con políticas que permitan dar a conocer la verdadera dimensión del problema" [58].

A este respecto, la falta efectiva de una soberanía digital y control jerarquizado dificulta la agilidad en la gestión de incidentes y su mitigación a nivel gubernamental[18]. No existen a día de hoy suficientes regulaciones que obliguen a los fabricantes, vendedores y distribuidores a competir en otros aspectos aparte de en la funcionalidad inmediata, quedando la responsabilidad relativa a la seguridad en el usuario final [59]. De igual modo, tampoco existen métodos generalistas que permitan proceder con esquemas de auditoría sencillos y factibles para abordar los diferentes niveles de seguridad ofrecidos por los dispositivos y servicios IoT. Además, entre dichos cuerpos reguladores existen reticencias fundadas ante la aplicación de este tipo de nuevas regulaciones de forma general, ya que podrían llegar a perjudicar el desarrollo tecnológico (y económico), al considerarse que un exceso regulatorio pueda acabar lastrando la productividad del mercado al frenar la innovación. Para ilustrar mejor este punto, conviene aclarar que una posible consecuencia de futuras certificaciones obligaría a ligar la certificación al uso que se plantea para el dispositivo, requiriendo de certificaciones nuevas si el uso planteado cambia por efectos de la innovación [60]. Algunos agentes reguladores, por tanto, tienden a actuar de forma conservadora para no lastrar el desarrollo económico, generándose vacíos regulatorios en los ecosistemas IoT que a largo plazo podrían perjudicar a la seguridad global.

Europa está liderando el desarrollo de iniciativas legislativas en el campo de las certificaciones de seguridad a través de distintos cuerpos reguladores [6]. Dichas normativas tratan de regular entre otros aspectos, los requisitos de seguridad de los dispositivos IoT en función de su uso y el tiempo de vida estimado de los algoritmos de seguridad utilizados. También pretenden definir conjuntos de buenas prácticas que garanticen, entre otros aspectos, la actualización de posibles vulnerabilidades sin intervención directa del usuario. Estos asuntos se están tratando también en Estados Unidos [61] [62], pero es en la Unión Europea en donde se lleva la iniciativa con regulaciones más avanzadas [13]. Para ello, organismos dependientes de la Comisión están abordando diversos aspectos de ciberseguridad para proponer una "regulación marco" al respecto. El progreso de estos trabajos sugiere que, en algunos casos, las tecnologías y dispositivos tengan que ser certificados externamente para verificar el cumplimiento del *Cybersecurity Act* [60]. Estas iniciativas en seguridad afectan a cualquier elemento *hardware*, dispositivos (individuales o en conjunto), técnicas o cualquier combinación de los mencionados elementos, según la definición de *Target of Evaluation* (TOE) por el *Common Criteria*[19] para la *IT*

[18] S. Lindemer, A. Khurshid, H. Wang, M. Furuhed, S. Raza, "Automating IoT Security: The EU CONCORDIA Approach", *IEEE IoT*, 2019, https://iot.ieee.org/newsletter/november-2019/automating-iot-security-the-eu-concordia-approach

[19] "Common Criteria" https://www.commoncriteriaportal.org/

Security Evaluation (ISO/IEC 15408)[20]. En particular, con el objetivo de conseguir hasta una calificación *Evaluation Assurance Level 7* (EAL7), la complejidad de los diseños debe tratar de reducirse en lo posible para que sea factible realizar análisis y testeos siguiendo un esquema "*white box*"[21], de manera que sea posible que actores independientes puedan confirmar los resultados en seguridad presentados por los desarrolladores y fabricantes [55].

1.3. OBJETIVOS DE LA INVESTIGACIÓN Y ESTRUCTURA DE LA TESIS

El objetivo del presente trabajo consiste en analizar la problemática existente en el campo de la seguridad en las comunicaciones en los dispositivos IoT, para elaborar una propuesta tecnológica que contribuya a reducir el número de problemas y mitigar sus efectos. Este ambicioso reto se plantea teniendo en consideración las distintas sensibilidades existentes en este campo (usuarios, empresas, mercados y cuerpos reguladores), para poder ofrecer una solución de compromiso que beneficie mutuamente a todos los actores involucrados para garantizar el éxito de la misma a medio y largo plazo. Una vez presentado el problema que se quiere abordar, esta sección detalla las metas perseguidas, que se han estructurado como respuestas a tres grupos de preguntas derivadas de la problemática expuesta.

El primer grupo de preguntas persigue analizar la viabilidad de un esquema de seguridad para los dispositivos que tenga un alcance lo más general posible.

- **C1:** ¿Se pueden abordar los problemas de seguridad de forma general, pero sin requerir necesariamente del uso obligatorio de tecnologías de seguridad/*framework* específicos por parte de los desarrolladores?
- **C2:** ¿Se puede diseñar un sistema de seguridad de manera que dispositivos IoT arbitrarios puedan incorporarse al mismo? ¿Qué requisitos mínimos serían necesarios?
- **C3:** ¿Cuál sería el esquema de seguridad técnico propuesto?

Un segundo bloque de preguntas persigue estudiar la viabilidad de un sistema de seguridad en el *gateway*/servidor que permita dar soporte al sistema definido para los dispositivos IoT, y que a su vez siga contribuyendo a solucionar los problemas enumerados en la anterior sección:

- **C4:** ¿Es factible integrar el esquema de seguridad en una infraestructura de comunicación horizontal que permita una comunicación de forma sencilla entre

[20] "Common Criteria for IT Security Evaluation. Part 1: Introduction and general model", 2017, https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R5.pdf

[21] Técnicas de *white-box testing*: Analizan la estructura y la lógica del programa. Estas técnicas se usan por lo tanto para ayudar en la monitorización del alcance del testeo y para guiar en la selección de los datos apropiados que servirán para conseguir que se ejecuten áreas de código que todavía no se habían ejecutado. Los casos de testeo generalmente se aplican al código que se pretende ejecutar. Sin embargo, estas técnicas también permiten encontrar errores simplemente examinando el código manualmente con herramientas de inspección de código y de asistencia de ejecución del mismo [174].

elementos de diferentes *verticals*?

- **C5:** ¿Cuál sería el esquema de seguridad técnico propuesto que habilitara una comunicación horizontal segura?
- **C6:** ¿Es posible diseñar un sistema de seguridad con las características anteriores que además permita la integración de otras tecnologías *software* IoT complementarias?
- **C7:** Este esquema de seguridad, ¿podría facilitar el desarrollo de algún sistema futuro de certificación más sencillo y automatizable?

Finalmente, un tercer grupo de preguntas persigue analizar en detalle la viabilidad técnica de las soluciones presentadas a través de experimentos llevados a cabo en laboratorio:

- **C8:** ¿Qué efectos tendría la aplicación de este esquema de seguridad en los dispositivos IoT frente a las soluciones de comunicación segura tradicionales?
- **C9:** ¿Qué desafíos y limitaciones afrontaría la aplicación de este esquema de seguridad en los *gateway*/servidores?
- **C10:** ¿Qué propuestas técnicas se pueden presentar para hacer frente a estas limitaciones?

En cuanto a la estructura de la tesis, el capítulo actual presenta una introducción que incluye la motivación y presentación del problema junto con los objetivos de la investigación.

El **capítulo 2** describe el marco teórico de la solución propuesta. Se presentan las principales características de los sistemas IoT y las tecnologías de seguridad existentes. De todas ellas se deducen sus alcances potenciales y principales problemáticas a resolver.

En el **capítulo 3** se presenta la solución propuesta por la tesis, es decir, IoTsafe. Inicialmente se lleva a cabo un análisis de algunos precursores reales de la misma basados en tecnología *Digital Signage* (DS). En la **sección 3.2** se presenta el concepto de desacoplo de seguridad de las aplicaciones, en el que se basa IoTsafe, y se valora su viabilidad, el esquema de seguridad propuesto y los requisitos técnicos, dando respuesta a las cuestiones **C1**, **C2** y **C3**. Tras presentar los aspectos fundamentales de IoTsafe, en la **sección 3.2.1** se valora también su viabilidad en sistemas RTOS.

La cuestión **C4,** relativa a los aspectos técnicos de la seguridad de las comunicaciones basada en proxificaciones en un servidor o Gateway, se resuelve más detalladamente en la **sección 3.2.2,** con la definición del contexto de seguridad usado por IoTsafe. Además, la cuestión **C5,** que persigue un esquema de las comunicaciones seguras de IoTsafe en el *gateway* y servidor, se responde en la **sección 3.2.3**, en la que se presenta IoTsafe como un *middleware* de seguridad. La **sección 3.3** describe un ejemplo completo de implementación IoTsafe, y responde a la cuestión **C6,** que persigue determinar el grado de compatibilidad que el esquema de seguridad propuesto tendría con otros desarrollos IoT. En la **sección 3.3** también se presenta de forma detallada una implementación genérica de IoTsafe que permite una integración abierta

y transparente de cualquier dispositivo que cumpla con dichos requisitos mínimos. La **sección 3.4** analiza el encaje de la solución dentro de las recomendaciones de la ITU y, finalmente, en la **sección 3.5** se enumeran las ventajas de IoTsafe. En esa sección se da respuesta en detalle a la cuestión **C7** relativa a las dificultades que encuentran las entidades certificadoras actuales encargadas de certificar el cumplimiento de las normativas de seguridad aplicables a IoT.

Al último grupo de cuestiones (**C8, C9 y C10**) se da respuesta por medio de diferentes pruebas experimentales llevadas a cabo a lo largo del desarrollo de la tesis. Así, en el **capítulo 4** se analizan los aspectos más importantes que tiene que superar IoTsafe para poder garantizar que, además de ser teóricamente una solución factible, resulte escalable en ecosistemas IoT reales. Para ello se estudia el efecto de la movilidad en las comunicaciones proxificadas en entornos con cobertura variable, la viabilidad de estas comunicaciones con protocolos de alto nivel y *hardware* de capacidades limitadas, su rendimiento y limitaciones en los *gateway* y servidores IoT, y los problemas de escalabilidad por las limitaciones inherentes al protocolo de comunicación segura que preferentemente utiliza IoTsafe.

Finalmente, el **capítulo 5** de la tesis ofrece las conclusiones de la investigación y presenta algunos trabajos futuros que se han planteado para continuar desarrollando la idea del desacoplo de la seguridad como nuevo paradigma alternativo de comunicación segura para los ecosistemas IoT.

2. MARCO TEÓRICO

Tal y como se ha mencionado en el capítulo anterior y se refleja en encuestas recientes [15] [16] [17], actualmente existe una tendencia mayoritaria en el uso de *hardware* y *software* de propósito general para el desarrollo de dispositivos y servicios IoT. Sus comunicaciones suelen estar basadas en los protocolos TLS/DTLS y por norma general, los despliegues de infraestructura IoT se enfocan a cubrir únicamente problemas concretos generando diferentes *verticals*. A continuación, se exponen de forma más detallada los aspectos de estas tendencias que más impacto tienen en la seguridad.

En la sección 2.1 se analizan algunas características comunes de Linux y de los RTOS que se utilizan en los sistemas IoT. La sección 2.2 describe los protocolos de seguridad más utilizados a nivel de Transporte en las implementaciones actuales de cualquier tipo. En la sección 2.3 se describen los aspectos más característicos del protocolo SSH que pueden contribuir a mejorar la seguridad. Finalmente, en 2.4 se estudia la diversidad en la arquitectura IoT y cómo esta realidad ahonda el problema de la seguridad, sobre todo ante la necesidad creciente de interconexión entre sistemas heterogeneos.

2.1. LINUX Y LOS RTOS: BASE DE LOS SISTEMAS IOT

Linux constituye el sistema operativo más extendido en el ámbito de los servidores de Internet. Además, su *kernel* incorpora compatibilidad para numerosos microcontroladores de alto nivel recientes [63] [64] que se están utilizando en un segmento considerable del IoT. Esto es debido a que el *hardware* de propósito general cada vez resulta más económico y energéticamente eficiente, como el basado en dispositivos *System on a Chip* (SoC) con procesadores ARM, o bien los dispositivos programables que emplean *Field-Programmable Gate Array* (FPGAs, Matriz de Puertas Lógicas Programable en Campo). Muchos de estos chips ya cuentan con estrategias muy avanzadas de administración de energía, que permiten integrar la potencia de sus procesadores en dispositivos con restricciones energéticas que deben funcionar con baterías por largos periodos de tiempo. Estas características invitan a los desarrolladores de dispositivos a utilizar *hardware* genérico para aplicaciones específicas, aunque sus prestaciones puedan resultar excesivas. A cambio, se acortan los tiempos de desarrollo y producción y se garantiza la posibilidad de actualización futura. Por esos mismos motivos, el desarrollo en estos dispositivos se realiza mayoritariamente en Linux [15]. Los dispositivos más sencillos y con más restricciones de energía apenas emplean ya soluciones *bare-metal* [17] prefiriendo alternativas basadas en RTOS. De entre todos los RTOS disponibles, el más popular es el FreeRTOS[22],

[22] "FreeRTOS™ - Real-time operating system for microcontrollers", https://www.freertos.org/

gracias a la gran variedad de *hardware* con el que es compatible, y además por contar con funcionalidades POSIX como Linux.

Linux engloba una familia de sistemas operativos de código abierto inspirados en los sistemas Unix. Las diferentes distribuciones de Linux se definen en torno a un componente fundamental denominado el *kernel*, de tipo monolítico con *preemptive multitasking*[23]. Esta característica diferencial, presente en la mayoría de sistemas operativos actuales, permite alternar la ejecución entre distintos procesos del sistema operativo, sin necesidad de que estos procesos hayan sido programados de forma específica para ello. La planificación de los recursos de CPU y la distribución de su tiempo entre los diferentes procesos es una tarea exclusiva del sistema operativo.

En sistemas operativos Linux, las aplicaciones funcionan en el espacio usuario, mientras que el *kernel* se ejecuta completamente en un espacio de memoria reservado (Fig. 2.1-1). La interacción entre ambos espacios se lleva a cabo mediante la Linux API.

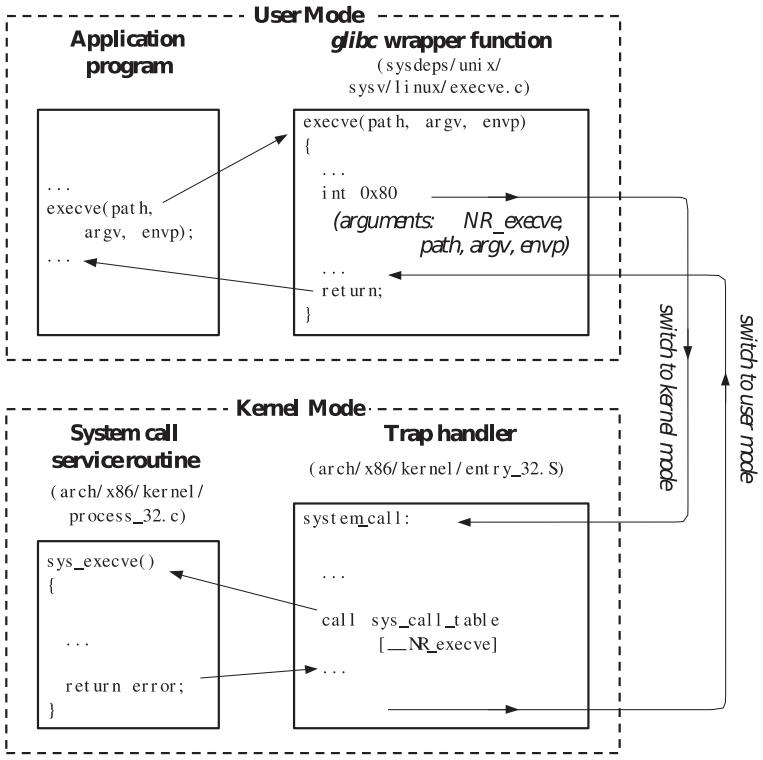


Fig. 2.1-1: Alternancia entre la ejecución de código entre el espacio usuario y el espacio kernel [65]**.**

[23] *Preemptive multitasking*: Capacidad multitarea de un sistema operativo que gestiona la asignación de los recursos para ejecución mediante un planificador (*scheduler*) en el *kernel.*

Esta API persigue ofrecer funcionalidades similares de portabilidad definidas en POSIX (Fig. 2.1-2) para que el código fuente de las aplicaciones y los binarios (gracias a la Linux ABI o *Aplication Binary Interface*) alcancen un alto grado de independencia del sistema operativo subyacente (Fig. 2.1-3) una vez cumplidas todas las dependencias de librerías compartidas.

El *kernel* ofrece un conjunto de llamadas al sistema que pueden utilizar las aplicaciones. Estas llamadas constituyen una interfaz muy estable en el tiempo ya que todas las llamadas al sistema reciben soporte indefinidamente[24]. El acceso a dichas llamadas al sistema se efectua normalmente a través de unas subrutinas que simplifican la labor y que ofrece la librería C estándar (*libc, glibc*, *eglibc*, *uClibc*, etc.). En un mismo sistema pueden coexistir múltiples librerías C de este tipo[25], permitiendo que operen programas basados en librerías diferentes. Esto facilita la distribución de ejecutables *stand-alone* que no tienen dependencias con ninguna librería del sistema y que pueden llegar a incluir también su librería C correspondiente. Si la aplicación no accede a las *system-calls* directamente, la dependencia con el *kernel* de dicha aplicación *stand-alone* es la que tenga la versión de librería C elegida. Además, Esta compatibilidad es perpetua para todas las versiones posteriores del kernel para el cual fue diseñado.

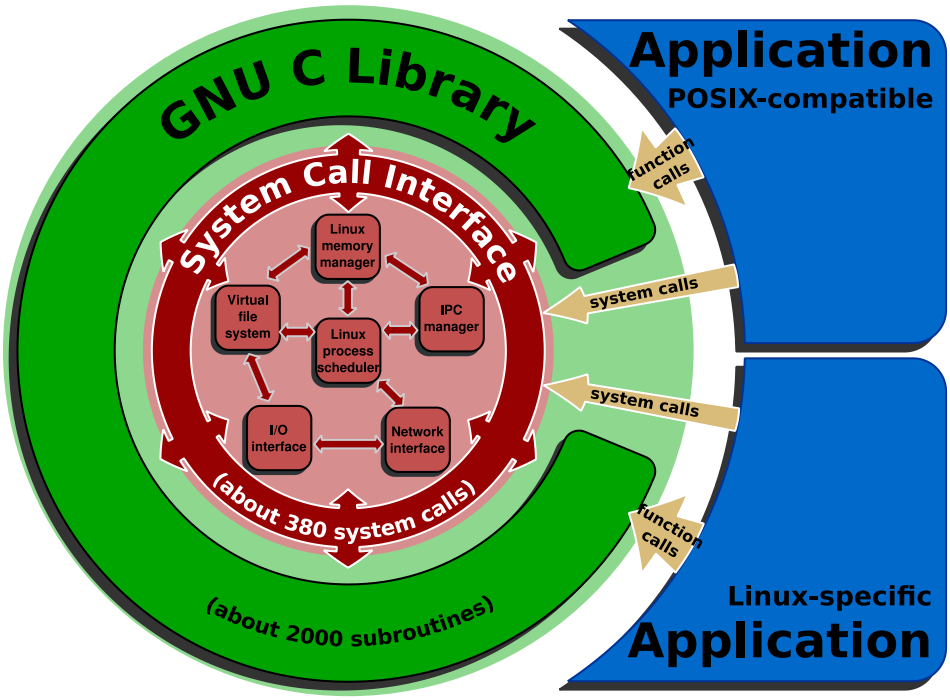


Fig. 2.1-2: GNU C Library como middleware (wrapper) del Linux kernel System Call Interface[26].

[24] "Adding a New System Call", *The Linux Kernel Documentation* https://www.kernel.org/doc/html/v4.10/process/adding-syscalls.html?highlight=system%20calls

[25] "ABI Policy and Guidelines - Appendix B. Porting and Maintenance", *The GNU C++ Library Manual* https://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html,

[26] "Linux kernel System Call Interface and glibc.svg", https://commons.wikimedia.org/wiki/File:Linux_kernel_System_Call_Interface_and_glibc.svg

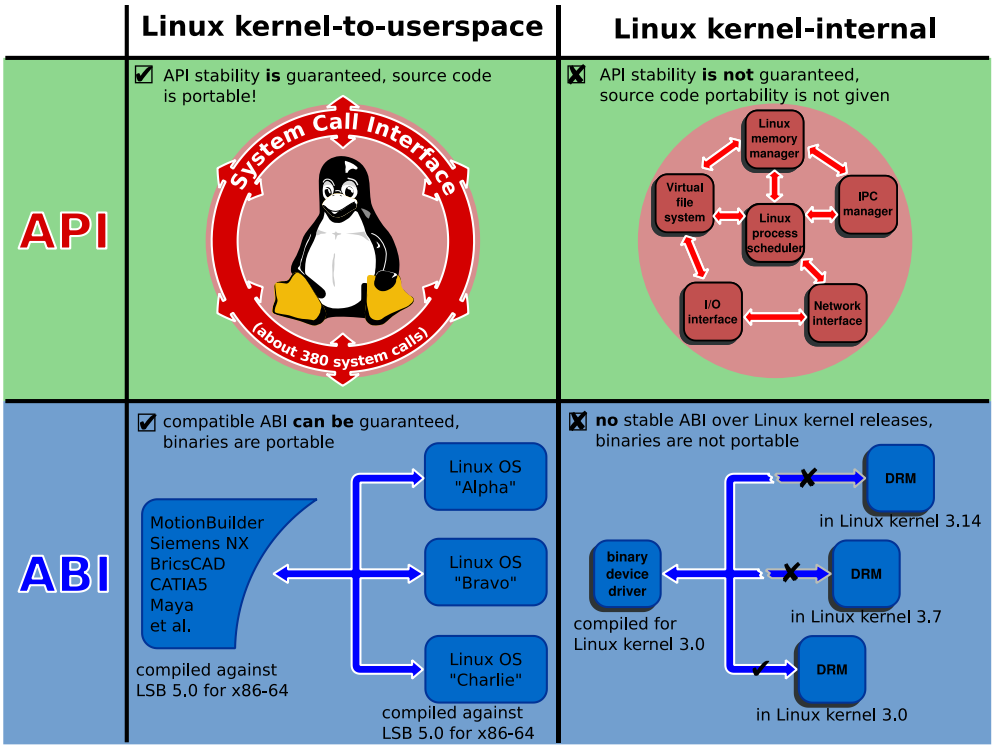


Fig. 2.1-3: Tabla de portabilidad de código Fuente en entornos Linux[27].

La compatibilidad de binarios entre distintas plataformas que comparten arquitectura sí está garantizada (una vez cumplidas las dependencias, o compilado como *stand-alone*). Por otro lado, la compatibilidad de módulos de *kernel* entre distintas versiones del *kernel* no está garantizada, ni a nivel binario ni de código fuente[28].

Por ese motivo, para el propósito del presente trabajo, se considera conveniente utilizar en los dispositivos IoT unos módulos *software stand-alone* que sean independientes del resto del *software*, que trabajen preferiblemente en el espacio usuario con apoyo de funciones muy concretas del espacio *kernel* mediante módulos del *kernel* estándar. De este modo se simplifican las actualizaciones, sin interferir con el resto del *software* del dispositivo.

Por otro lado, por lo que respecta a la estructura de los RTOS, ésta resulta bastante distinta a la de Linux, reduciéndose notablemente el tamaño del *kernel* y del sistema a unos pocos Kbytes. En el caso de FreeRTOS, además de operar con un *microkernel*, el arranque del sistema carga un único binario que el microcontrolador utiliza de forma estática y que ya contiene el sistema operativo junto a las aplicaciones. No existe, por tanto, portabilidad binaria como se describe en Linux, pero se han desarrollado estrategias que permiten la actualización *en caliente* de dicho binario de forma remota[29], modificando el sistema de arranque del microcontrolador.

[27] "Linux Kernel Interfaces", https://commons.wikimedia.org/wiki/File:Linux_kernel_interfaces.svg
[28] https://kernel-team.pages.debian.net/kernel-handbook/ch-versions.html
[29] Benjamin Bucklin Brown, "Over-the-Air (OTA) Updates in Embedded Microcontroller Applications: Design Trade-Offs and Lessons Learned" *Analog Dialogue*, vol. 52, noviembre de 2018, https://www.analog.com/en/analog-dialogue/articles/over-the-air-ota-updates-in-embedded-microcontroller-applications.html

2.1.1. Seguridad en los servidores IoT

Los sistemas operativos POSIX de tipo UNIX basan su seguridad, no sólo en las funcionalidades ofrecidas por el sistema operativo en sí, sino también en un conjunto de buenas prácticas que deben seguir los usuarios y administradores del sistema. De forma nativa, los sistemas UNIX funcionan con sistemas de archivos y descriptores con bits de permisos y propietario, para configurar los distintos tipos de acceso para usuarios y grupos. Estas prestaciones POSIX de Linux permiten el funcionamiento de privilegios separados por usuarios y procesos. La separación de privilegios puede llegar a extenderse al dominio local de red (dentro de la máquina), gracias a que Linux cuenta con Netfilter (Fig. 2.1-4), un potente *stateful packet filter* (filtro de paquetes con estado) que actúa como *firewall* y puede ser configurado desde el espacio usuario por medio de la herramienta iptables.

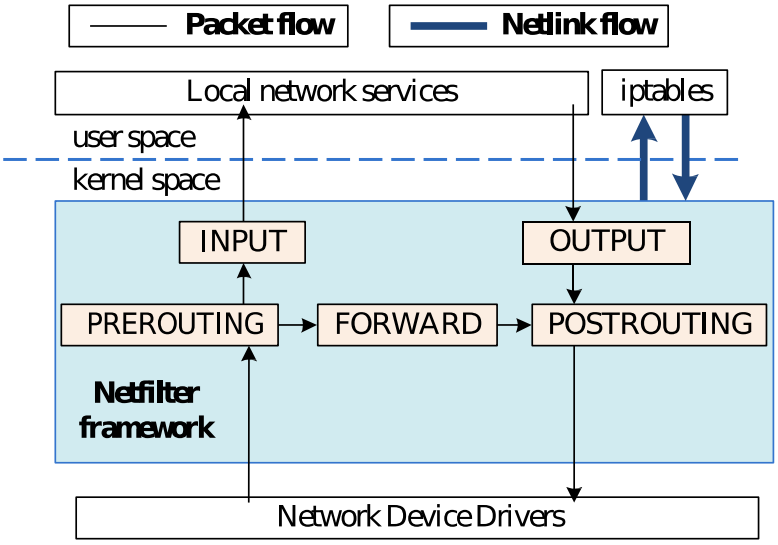


Fig. 2.1-4: Diagrama de funcionamiento de Linux Netfilter [64]**.**

El Sistema operativo Linux, dispone del *framework* LSM que permite la carga de diversos tipos de módulos de seguridad, que operan por medio de *hooks* en el *kernel,* permitiendo a estos módulos realizar llamadas al sistema ante acciones importantes que el *kernel* deba procesar [66].

Los módulos de seguridad que utilizan dicho framework, como *SeLINUX* o *AppArmor*, buscan principalmente proteger los accesos a recursos mediante políticas de ejecución (*SeLinux*) o perfiles de configuración de la seguridad para cada aplicación (*AppArmor*), evitando que aplicaciones en ejecución utilicen alguna vulnerabilidad o error de configuración para escalar privilegios o generar cualquier otro tipo de daño. Este tipo de escalado de privilegios se produce apovechando alguna vulnerabilidad oculta del kernel [67], o cuando se han aplicado de forma incorrecta conjuntos de permisos a archivos o carpetas sensibles, que pueden ser explotados para realizar un ataque. La configuración incorrecta de servicios del sistema también puede favorecer el éxito de ataques de este tipo. Un sistema Linux bien estructurado y sin

vulnerabilidades no debería tener problemas de seguridad. No obstante, dado que determinadas configuraciones vulnerables pueden pasar desapercibidas, la NSA recomienda [68] el uso de módulos de seguridad, como *SeLinux,* con perfiles de configuración estándar, que sirven como doble protección ante problemas que hayan podido pasar desapercibidos.

En todo caso, cualquier esquema que utilice la LSM API del *kernel* sirve para incrementar la seguridad, contribuyendo a mitigar posibles vulnerabilidades desconocidas derivadas de una pobre planificación de seguridad, configuraciones inseguras o puntos vulnerables del sistema que se hayan pasado por alto (servicios obsoletos vulnerables, *password* por defecto, etc.) que pudiesen ser utilizados de manera ilícita, y en particular para escalar privilegios [69].

2.1.2. **Seguridad en los dispositivos IoT**

Los dispositivos IoT normalmente tienen funciones específicas. Los aparatos más complejos, utilizan generalmente *hardware* de alto nivel, como ocurre con las cámaras de seguridad o dispositivos avanzados de control de acceso. Por el contrario, los dispositivos IoT más sencillos, pueden diseñarse a partir de microcontroladores simples con unos pocos Kbytes de memoria RAM y ROM [70]. Estos dispositivos no requieren de complejas medidas de seguridad internas, ya que las aplicaciones suelen ser sencillas. Para la implementación de seguridad en las comunicaciones en estos casos, bastaría con controlar adecuadamente el tráfico con el exterior del dispositivo mediante sencillas reglas estáticas de firewall. Esto se aplica igualmente a dispositivos Linux sencillos que únicamente utilicen aplicaciones del mismo estilo.

2.2. TLS Y DTLS: PROTOCOLOS PARA COMUNICACIÓN SEGURA

Actualmente, puede decirse que el estándar abierto *de facto* para seguridad en las comunicaciones, a nivel de Transporte, en dispositivos de alto nivel es *Transport Layer Security* (TLS) [22] funcionando sobre TCP. De igual modo puede considerarse también como estándar su versión ligera para comunicaciones sobre UDP, el *Datagram Transport Layer Security* (DTLS) [23]. Estos protocolos combinan una estrategia de cifrado asimétrica para autenticar la contraparte por medio de certificados X.509, con un cifrado simétrico para proteger las comunicaciones. Tanto TLS como DTLS se utilizan de forma generalizada en las comunicaciones de servicios web (HTTPS), email, VoIP, mensajería instantánea, etc. [22].

TLS [21] integra una colección de protocolos y algoritmos de seguridad que definen los procedimientos de autenticación, establecimiento y cifrado de las comunicaciones. Cada versión de TLS engloba varios de estos algoritmos, los cuales se negocian en el momento del establecimiento de la conexión. Si alguno de estos algoritmos se vuelve potencialmente inseguro, se sustituye por otro más robusto en la siguiente versión de TLS. El procedimiento de negociación de estos algoritmos también es objeto de revisión constante.

El establecimiento de la conexión entre cliente y servidor al utilizar TLS se realiza siempre por iniciativa del cliente. TLS proporciona una protección criptográfica del canal de comunicaciones para evitar ataques de seguridad en algún punto intermedio evitando, entre otros, que sea interceptada (*eavesdropping*) o pueda alterarse (*tamptering*) de forma arbitraria o a través de un agente usurpador de la identidad (*Man In The Middle*, MITM).

2.2.1. Procedimiento de conexión con TLS

El procedimiento de *handshake* incluye los siguientes pasos [21]:

1) El cliente se conecta a un servidor TLS, solicitando una comunicación segura. Para ello proporciona una lista de funciones de cifrado y *hash* que soporta.
2) De la lista suministrada, el servidor elije un *cipher* (función de cifrado) y un *hash* que también soporta, y comunica al cliente su elección.
3) El servidor transmite al cliente los datos públicos relativos a su certificado, con los que el cliente puede comprobar y validar la identidad del servidor. Dicho certificado contiene la información relativa al servidor (en particular su nombre, dominio y dirección IP), la entidad certificadora (CA) que avala la autenticidad del certificado, y la clave pública de cifrado del servidor.
4) El cliente confirma la validez del certificado.
5) Se utiliza un algoritmo de intercambio de claves para generar una única clave de cifrado de sesión utilizando la infraestructura de clave pública o PKI (*Public Key Infrastructure*), de la que forma parte la clave pública del servidor.

2.2.2. Problemática de TLS/DTLS en IoT

TLS realiza la autenticación por parte del servidor empleando certificados reconocidos por una autoridad competente (*Certification Authority*). Sin embargo, la autenticación del cliente, aunque es posible[30,31] [71], es una tarea que normalmente se delega a la aplicación que hace uso del protocolo. En cambio, otros protocolos de seguridad como IKEv2 [72] (*Internet Exchange Protocol*) usado por IPSec (*IP Security*) [73] o SSH [74], sí están diseñados para proporcionar por defecto esta funcionalidad.

El uso tradicional que se ha dado a TLS, ha sido el de proteger las comunicaciones cliente/servidor en las que únicamente es relevante autenticar al servidor al inicio de las mismas. Estas comunicaciones se caracterizan por:

- Estar orientadas a una arquitectura cliente/servidor.

[30] Dani Grant, "Introducing TLS with Client Authentication", *The Cloudflare Blog*, 2017, https://blog.cloudflare.com/introducing-tls-client-auth/
[31] IBM Corporation, "How SSL and TLS provide identification, authentication, confidenciality, and integrity" *IBM Knowledge Center*, 2019, https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10670_.htm

- Requerir de una autoridad certificadora para los servidores.
- La necesidad de disponer de un dominio público por parte de los servidores de manera que éste quede ligado al certificado.

Las características arriba mencionadas llevan a descartar la posibilidad de utilizar ampliamente TLS o DTLS para comunicar dispositivos IoT, si estos no pueden realizar una validación completa de la autoridad certificadora. Obviar o simplificar este paso podría permitir ataques de tipo *man-in-the-middle,* pues entidades que suplantaran la identidad de fuentes legítimas podrían emplear certificados fraudulentos reconocidos como válidos por los dispositivos IoT, al no corroborarlos con las autoridades de certificación [75] [76]. En el caso de no poder realizarse esta validación, es imprescindible implementar algún tipo de autenticación complementaria [77]. Por otro lado, la comunicación entre dispositivos IoT de forma directa empleando TLS no resulta tan inmediata, ya que no suelen contar con certificados válidos, siendo en todo caso auto-emitidos y favoreciendo la usurpación de identidad de un dispositivo por un agente externo.

Por lo que respecta al desarrollo de aplicaciones, compilar una aplicación para utilizar TLS puede requerir del uso de las primitivas nativas del propio *framework* de TLS, o el uso de un *framework* más completo que incluya TLS. Esta es la situación de la plataforma .NET de Microsoft, en donde la versión de TLS utilizada depende principalmente de la versión de .NET *Framework*[32] que se esté empleando a la hora de compilar el *software*. En el caso de aplicaciones basadas en Android, la versión TLS depende principalmente de la versión del sistema operativo utilizado o de la manera en la que se haya desarrollado la aplicación [33].

En estas dos circunstancias, para llevar a cabo una actualización efectiva de TLS en entornos .NET del programa a desarrollar, se deberá compilar con una versión posterior del *framework* completo. En el caso de Android, se deberá volvier a compilar la aplicación para una versión más moderna de Android, y actualizar el sistema operativo en el dispositivo [78]. Si la aplicación emplea su propia versión del *framework* TLS, deberá sustituirla, adecuarse a las primitivas de la nueva versión del *framework* y recompilarse.

Sin embargo, pese a esas nuevas facilidades de Android, un estudio estadístico [33] realizado de 2015 a 2017 confirma la lentitud de los desarrolladores a la hora de abordar de forma efectiva soluciones frente a problemas de seguridad. En dicho estudio, por ejemplo, se constata que se necesitó más de año y medio para que sólo el 50% de aplicaciones dejara de utilizar los cifrados RC4 de TLS catalogados como inseguros tras la pulicación de su vulnerabilidad [79] (Fig. 2.2-1).

[32] "Transport Layer Security (TLS) best practices with the .NET Framework", *Programación para redes en .NET Framework*, 2018, https://docs.microsoft.com/en-us/dotnet/*framework*/network-programming/tls

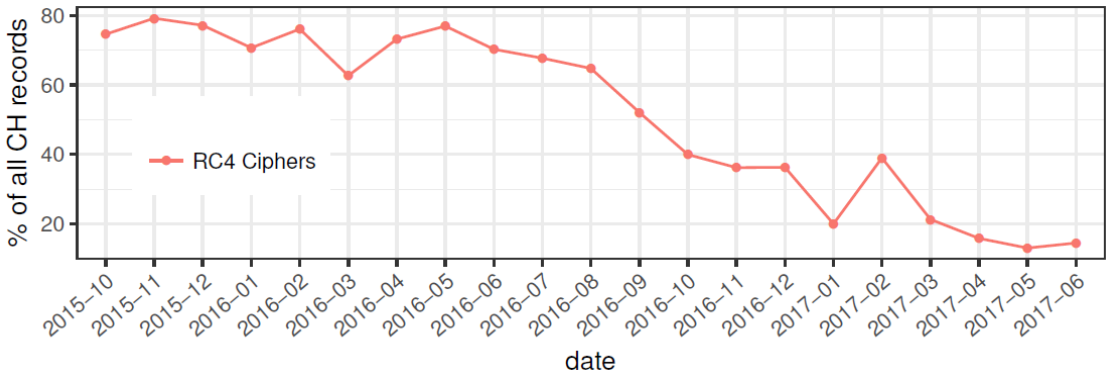


Fig. 2.2-1: Porcentaje de aplicaciones detectadas utilizando algoritmos de cifrado RC4 tras la publicación de vulnerabilidades que los afectaban en Febrero de 2015 [33]***.***

Esta circunstancia se puede explicar si se tiene en cuenta que cualquier tipo de actualización de estos *framework* o del propio TLS (si este se usa de forma nativa en la programación del código fuente) puede requerir de cambios, de mayor o menor calado, en la programación interna del *software*, complicando el soporte técnico para la aplicación en el medio y largo plazo (Fig. 2.2-2).

Fig. 2.2-2: Problemas de actualización de TLS: La aplicación que hace uso del framework debe adaptarse a las nuevas primitivas del framework y recompilarse. No es posible una simple "actualización de librerías TLS".

2.3. *SECURE SOCKET SHELL* (SSH): EXTENSIÓN DE LA SEGURIDAD LINUX ENTRE DISPOSITIVOS

Linux y los RTOS de tipo POSIX son capaces de implementar una separación de privilegios entre usuarios, y por extensión, entre procesos diferentes. Dichos procesos pueden comunicarse entre sí de forma transparente por medio de *socket* locales de red, pudiendo conseguir *de facto* un desacoplo de seguridad exitoso entre ellos, con determinadas configuraciones locales. Sin embargo, es necesario extender dicho desacoplo de seguridad a las comunicaciones entre dispositivos, para poder disponer de un esquema de comunicación completo. Para ello, en este trabajo se presenta como alternativa el uso de un protocolo de comunicación segura capaz de funcionar *stand-alone*: SSH.

SSH surge inicialmente de la necesidad de ofrecer una alternativa segura al método estándar existente previamente para establecer interfaces de terminal de sistema operativo entre dispositivos conectados por red. Este método lo implementaba el protocolo *TELetipe NETwork* (TELNET), cuyo desarrollo inicial data de 1969. Fue uno de los primeros en surgir [80] y

evolucionó progresivamente ofreciendo nuevos servicios [81]. Además de TELNET, fueron surgieron otros protocolos que dieron lugar a los servicios ofrecidos por los *r_commands* de BSD [82] para automatizar determinados procesos remotos a nivel de sistema operativo sin necesidad de autenticarse constantemente. Sin embargo, al igual que TELNET, la mayoría de estos protocolos tenían graves problemas de seguridad derivados, entre otros aspectos, del envío en claro de las credenciales de autenticación (Fig. 2.3-1).

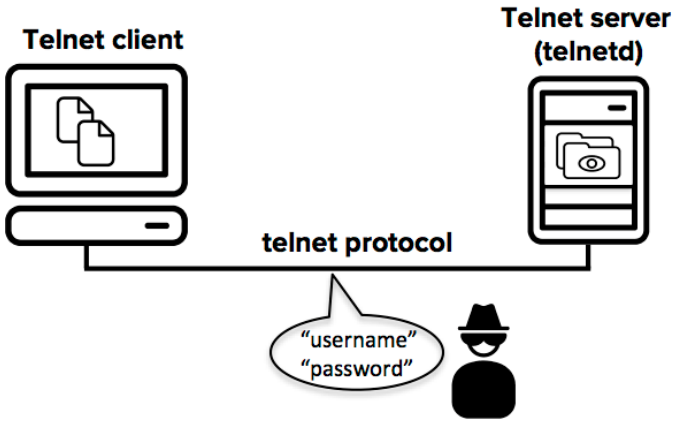


Fig. 2.3-1: Vulnerabilidad de TELNET ante eavesdropping[33].

SSH es un protocolo de Internet que puede ofrecer distintos servicios de seguridad como la autenticación, integridad y cifrado (Fig. 2.3-2). También permite utilizar diferentes servicios seguros como el de *login* remoto, ejecución remota y transferencia de ficheros; todo en dispositivos a los que se accede a través de redes inseguras (Fig. 2.3-3).

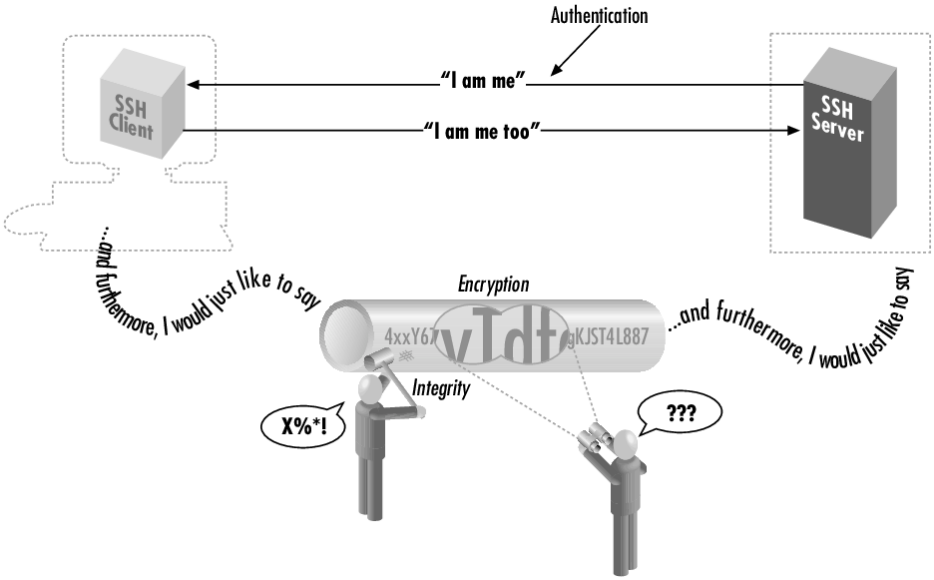


Fig. 2.3-2: Protección ofrecida por SSH: Autenticación, cifrado e integridad en los datos[34].

[33] "Telnet and SSH as a secure alternative", *SSH.com* , https://www.ssh.com/ssh/telnet
[34] Daniel J. Barrett and Richard E. Silverman, "The SSH Protocol" *The Secure Shell: the definitive guide,* 2001, https://nnc3.com/mags/Networking2/ssh/ch01_03.htm

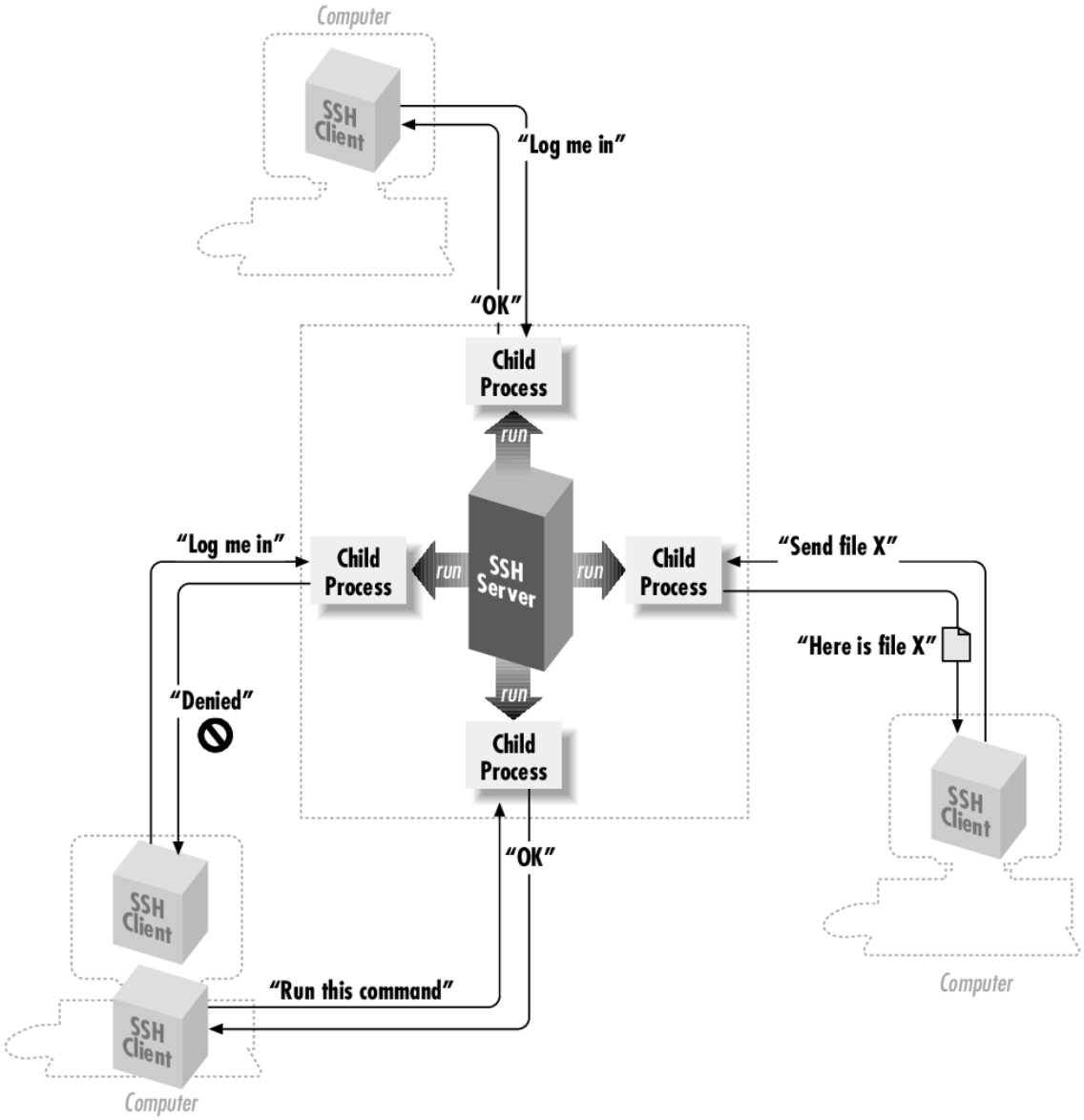


Fig. 2.3-3: Servicios seguros básicos ofrecidos por SSH[35].

Este protocolo, aunque puede incorporarse nativamente en cualquier aplicación o servicio para las comunicaciones seguras por medio de librerías, suele emplearse como protocolo de aplicación que da servicio a otras aplicaciones. Dada la complejidad y diversidad de sus servicios, el protocolo divide las funciones necesarias en tres grandes módulos: *Transport Layer Protocol*, *User Authentication Protocol* y *Connection Protocol*. (Fig. 2.3-4).

[35] Daniel J. Barrett and Richard E. Silverman, "What is SSH?" *The Secure Shell: the definitive guide,* 2001, https://nnc3.com/mags/Networking2/ssh/ch01_01.htm

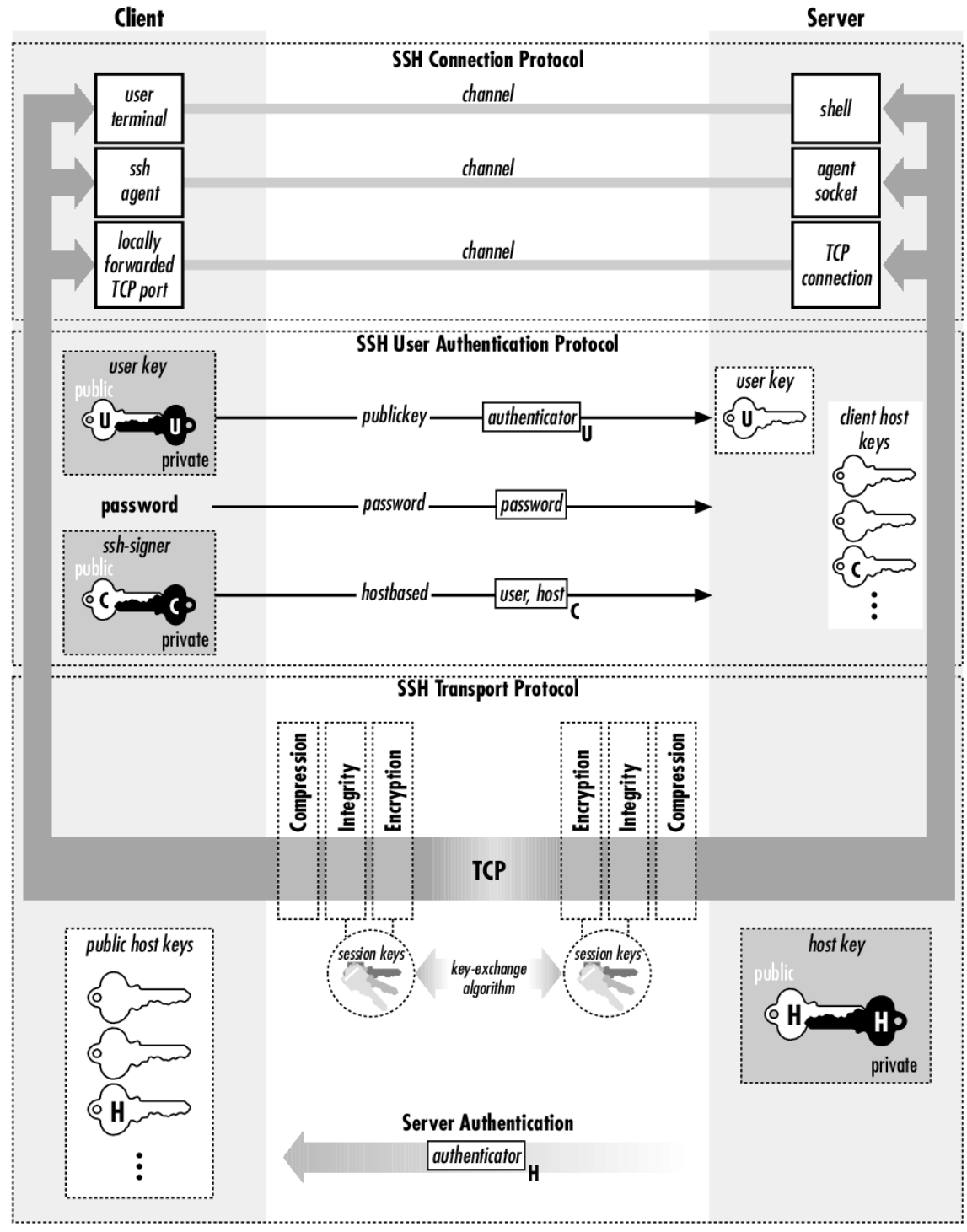


Fig. 2.3-4: Arquitectura de SSH (versión 2)[36].

[36] Daniel J. Barrett and Richard E. Silverman, "Inside SSH-2" *The Secure Shell: the definitive guide,* 2001, https://nnc3.com/mags/Networking2/ssh/ch03_05.htm

Los módulos de transporte [83] y autenticación [74] utilizan algoritmos y métodos equivalentes a los empleados por TLS en el establecimiento de las comunicaciones entre las partes. El módulo que desarrolla el *Connection Protocol* [84] establece los servicios de comunicación característicos que definen a SSH entre los que se encuentra el *port-forwarding*.

SSH utiliza canales para transmitir la información entre el cliente y el servidor. Estos canales son de cuatro tipos [85], dependiendo de la clase de servicio ofrecido:

1) ***Session*:** Canal utilizado por el cliente y el servidor para intercambiar mensajes relativos al protocolo SSH y al *Shell* interactivo.
2) **X11:** Canal empleado para transmitir de forma segura la información entre un servidor X y un cliente.
3) ***Direct-tcpip***: Canal invocado por el cliente hacia el servidor cuando se recibe una conexión en una proxificación directa (también conocida como *direct port forwarding* y *local port forwarding*).
4) ***Forwarded-tcpip*:** Canal invocado por el servidor hacia el cliente cuando éste recibe una conexión en una proxificación reversa (también conocida como *reverse port forwarding* o *remote port forwarding*).

Una vez se establecen las instancias de los canales correspondientes, sus comunicaciones respectivas pueden comenzar, tal y como se muestra en la Fig. 2.3-5

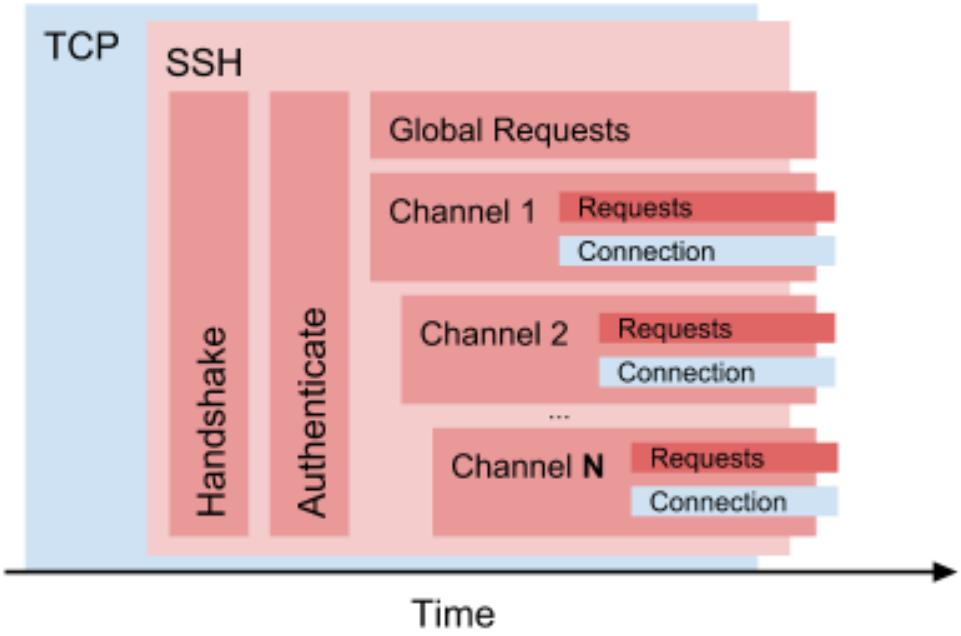


Fig. 2.3-5: Diagrama de secuencia de eventos en la comunicación por SSH a través de canales[37].

[37] "What is the Secure Shell protocol?", *Gopher Academy Online Learning,* 2015, https://github.com/gopheracademy/gopheracademy-web/blob/master/content/go-and-ssh.md

SSH funciona sobre TCP y lo mismo ocurre con sus servicios derivados. El servicio de *port-forwarding*, por lo tanto, sólo puede ofrecerse directamente a aplicaciones que trabajen con TCP. Todos los servicios de SSH se ofrecen a través de canales de comunicación que se multiplexan en la sesión establecida entre el cliente y servidor. Entre los servicios ofrecidos, existen dos de tipo *port-forwarding*:

1) ***Direct port-forwarding*:** El primer caso de *port-forwarding* se encuentra en aquellos escenarios en los que la parte responsable del establecimiento de la sesión SSH requiere que un *socket* de un servicio existente de la parte remota se ponga a disposición local (Fig. 2.3-6).

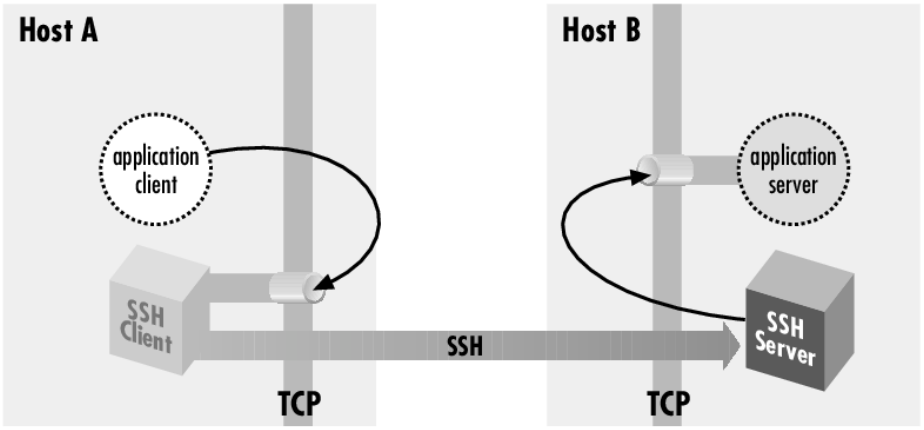


Fig. 2.3-6: Ejemplo de direct port-forwarding[38].

2) ***Reverse port-forwarding*:** Este segundo caso es opuesto al primero. En este escenario, la parte responsable del establecimiento de la sesión SSH pone un socket abierto de algún servicio local existente, a disposición de la parte remota, y de forma local en el dispositivo remoto (Fig. 2.3-7).

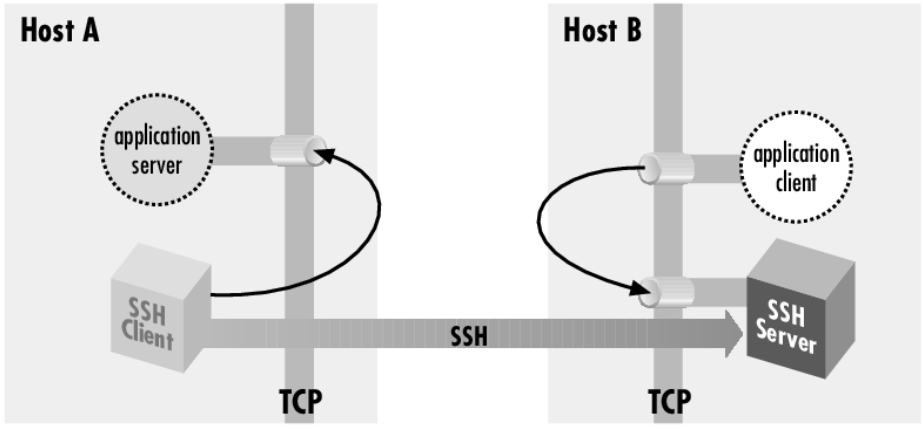


Fig. 2.3-7: Ejemplo de reverse port-forwarding[38].

[38] Daniel J. Barrett and Richard E. Silverman, "Port Forwarding" *The Secure Shell: the definitive guide,* 2001, https://nnc3.com/mags/Networking2/ssh/ch09_02.htm

El *socket* que comunica el servicio *port-forwarding* no tiene que ser necesariamente un *socket* local. Este *socket* puede pertenecer a un servicio remoto (de una máquina diferente). En estos casos, el tráfico no protegido por la sesión SSH se extiende más allá del tráfico local en dispositivo origen y/o destino.

2.3.1. Diferencias entre un servicio proxy general y el servicio de proxificación de SSH

Existen algunas diferencias entre el *port-forwarding* y el funcionamiento de un *proxy* estándar que conviene poner de manifiesto. Para ello, se procede a continuación a definir brevemente algunos conceptos que se emplearán en los ejemplos de esta sección:

- Las comunicaciones indicadas con flechas en las figuras de esta sección pueden ser bidireccionales, aunque la flecha sea unidireccional. El sentido de la flecha marca el origen de la comunicación y el destino de la misma para indicar qué actor es el que origina el proceso de comunicación.
- En los ejemplos se especifican características de las direcciones IP y el valor de los puertos de los *socket* de red involucrados:
  - **PrivateIP**: Hace referencia a una IP no pública que puede ser de red local, por ejemplo.
  - **Rand (Puerto)**: Hace referencia a un valor no conocido, asignado por el sistema operativo para los puertos normalmente origen de las comunicaciones.
  - **$Port_A$**: Puerto de destino final de la solicitud de comunicación de la aplicación.
  - **PublicIP**: Dirección IP pública accesible desde Internet.
  - **$Port_{A'}$**: Proxificación del puerto de destino $Port_A$, que normalmente ocurre en una máquina distinta a la que alberga el $Port_A$ original.
  - **127.0.0.1**: Dirección IP local de un dispositivo para referirse a sí mismo. Cuando el origen y/o destino de una comunicación tiene esta dirección, el tráfico no fluye fuera del dispositivo.

Para discernir más claramente la diferencia entre un *proxy* estándar y el servicio de proxificación de SSH, se incluye a continuación una descripción del funcionamiento general de un *proxy*. Un *proxy* (servidor) se define normalmente como un dispositivo o aplicación que figura como intermediario de las peticiones de un cliente a un servidor. Un ejemplo de este caso se presenta en la Fig. 2.3-8.

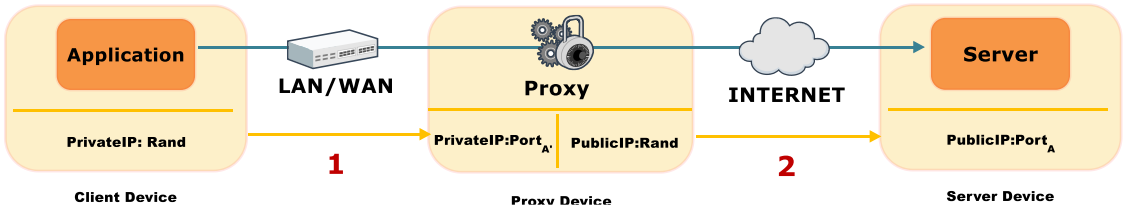


Fig. 2.3-8: Funcionamiento general de un Proxy.

La acción de *port-forwarding* que implementa SSH también permite acceder a recursos remotos, pero de forma diferente y más general que el *proxy*:

1) SSH requiere una sesión entre el cliente y el servidor SSH. La sesión debe estar además cifrada y requiere de credenciales del cliente para poderse establecer. Un *proxy* normalmente se ejecuta en una misma máquina y ofrece su servicio entre distintas interfaces, mientras que el servicio de SSH puede funcionar entre máquinas distintas.
2) Las comunicaciones cursadas por *port-forwarding* están protegidas por la sesión y su cifrado.
3) Existe la posibilidad de *port-forwarding* reverso que, a diferencia del directo, permite ofrecer recursos del cliente al servidor. Esto es posible en los *proxy*, aunque resulta menos común. Se denomina proxificación reversa y suele aplicarse en servidores *proxy* web para mejorar el funcionamiento de HTTP[39].

A continuación se presenta un ejemplo de comunicación por *port-forwarding* directo (Fig. 2.3-9). En este primer caso, la aplicación accede a un recurso remoto ofrecido por un servidor en un puerto $Port_A$. Este caso puede parecer inicialmente muy similar al del *proxy* estándar. Sin embargo, las comunicaciones se protegen y cifran de forma transparente entre el dispositivo cliente y el dispositivo servidor (2). Además, la comunicación entre el servidor SSH y la aplicación Servidor (3) se produce de forma local, al igual que la solicitud presentada por la Aplicación al Cliente SSH (1).

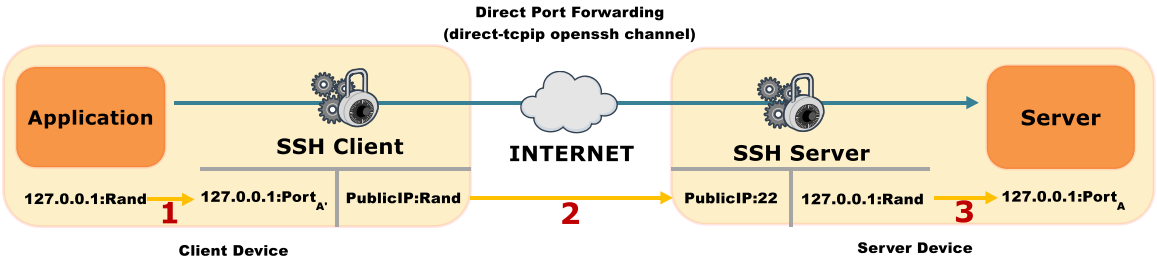


Fig. 2.3-9: Direct Port Forwarding: Ejemplo en el que una aplicación accede a un recurso remoto.

El procedimiento de comunicación por proxificación directa sería el siguiente:

- Inicialmente el cliente SSH establece una sesión con el servidor SSH (2). Esta sesión tiene como objeto preparar la proxificación directa por la que posteriormente la aplicación puede conectarse con el servidor.
- Tras quedar establecida la sesión SSH, en el dispositivo cliente (*Client Device*) se establece la proxificación directa en forma de *socket* (127.0.0.1:$Port_{A'}$) que conduce de forma transparente al *socket* destino en el servidor (127.0.0.1:$Port_A$).
- Cuando la aplicación desea comunicarse con el servidor, puede hacerlo conectándose al *socket* local (1), y el servidor SSH se conecta en su nombre (3) con el servidor y de

[39] "What Is a Reverse Proxy Server?", *NGINX glossary*, https://www.nginx.com/resources/glossary/reverse-proxy-server/

forma remota. Las respuestas se conducen de manera transparente hacia la aplicación, permitiendo así una comunicación bidireccional transparente.

La estrategia de *port-forwarding* ofrece seguridad en las comunicaciones que cursan redes no seguras o atraviesan Internet. Estas comunicaciones no son tuneladas por la sesión SSH a través de la proxificación, como podría ocurrir con una *Virtual Private Network* VPN de nivel 3 o de nivel 2. En una tunelación se encapsulan paquetes o tramas completas de tráfico, y se conducen al otro extremo del túnel siguiendo reglas de enrutamiento. Las comunicaciones proxificadas entre la aplicación y el servidor, (una vez establecida sesión SSH habilitante) se realizan del siguiente modo:

- Las comunicaciones en el segmento (1) se producen entre dos procesos del sistema (la aplicación y el cliente SSH). En esa comunicación, el cliente SSH recibe el *payload* del protocolo de Transporte originado en la aplicación. Este *payload* incluye los datos de la aplicación manejados en el protocolo de aplicación con su formato correspondiente.
- El cliente SSH recibe este *payload* y lo codifica dentro de un canal SSH, que se multiplexa a su vez con el resto de canales y se envía cifrado hacia el destino (Fig. 2.3-5).
- El servidor SSH recibe el *payload* e inicia una comunicación con el *socket* destino (3), totalmente independiente del segmento (1), de manera que, a ojos de la aplicación, el servidor destino se encuentra localmente dentro del dispositivo cliente y a ojos del servidor, la Aplicación que contacta con él se encuentra localmente dentro del dispositivo servidor.

Las comunicaciones cursadas a través de proxificaciones SSH quedan protegidas en su tránsito por Internet (2). Sin embargo, la comunicación en los segmentos (1) y (3) sigue estando desprotegida y a merced de otros procesos locales de ambos dispositivos (cliente y servidor).

La comunicación por *port-forwarding* reverso constituye una nueva funcionalidad que los servidores *proxy* estándar normalmente no presentan. Por ejemplo, la proxificación reversa utilizada en algunos servidores web, opera conjuntamente con las proxificaciones directas y se emplea como complemento de estas últimas para mejorar, entre otros aspectos, la compatibilidad del proxy web con desarrollos web que necesitan de una alta integración el explorador web y las variables de sesión establecidas.

La proxificación reversa en SSH consiste en ofrecer un servicio o recurso del $Port_A$ de la aplicación del dispositivo cliente, en el dispositivo servidor. En este caso, al igual que en el caso anterior, la sesión la establece el cliente hacia el servidor SSH (2), pero con el objeto de que el cliente ponga un recurso de la aplicación (127.0.0.1:$Port_A$) a disposición del servidor en forma de proxificación (127.0.0.1:$Port_{A'}$). En el caso anterior, la proxificación directa tenía una misión diferente, ya que su objeto era traer un recurso del servidor hacia el cliente. De este modo, a

través de una proxificación reversa la aplicación del dispositivo servidor puede acceder al recurso ofrecido por la aplicación del dispositivo cliente, pero de forma local a través del puerto $Port_{A'}$ (Fig. 2.3-10).

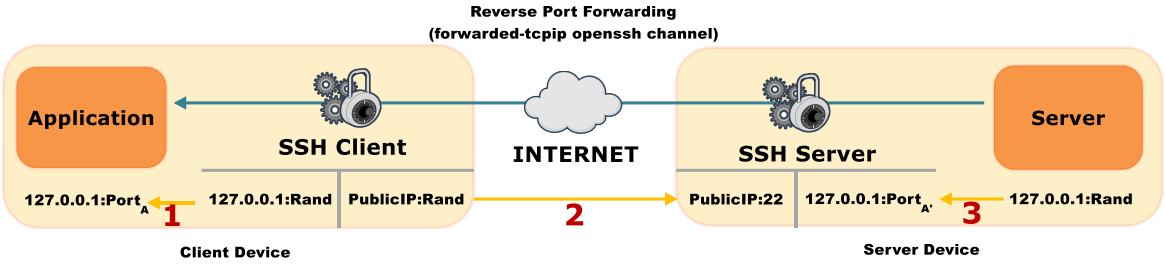


Fig. 2.3-10: Reverse Port Forwarding: Ejemplo en el que una aplicación ofrece un recurso a un servidor remoto.

El procedimiento de comunicación por proxificación reversa, una vez establecida, es el siguiente:

- Las comunicaciones en el segmento (3) se producen entre dos procesos del sistema (la aplicación y el servidor SSH). En esa comunicación, el servidor SSH recibe el *payload* del protocolo de transporte originado en la aplicación. Este *payload* incluye los datos de la aplicación manejados en el protocolo de aplicación con su formato correspondiente.
- El servidor SSH recibe este *payload*, y lo codifica dentro de un canal SSH que se multiplexa a su vez con el resto de canales y se envía cifrado hacia el dispositivo cliente.
- El cliente SSH, recibe el *payload* e inicia una comunicación con el *socket* destino (1) totalmente independiente del segmento (3), de manera que, a ojos de la aplicación del servidor, la aplicación del cliente que ofrece el recurso se encuentra localmente dentro del dispositivo servidor y, a ojos del cliente, la aplicación que contacta con él se encuentra localmente dentro del dispositivo cliente.

Al igual que en el caso anterior, sólo las comunicaciones entre el dispositivo servidor y cliente (2) se encuentran cifradas. Los segmentos 1 y 3 no reciben ningún tipo de protección extra y se encuentran a merced de procesos locales de sus dispositivos respectivos.

En los dos ejemplos anteriores, el servidor SSH y la aplicación/servidor con la que se comunica en el segmento 3 comparten dispositivo: *el dispositivo servidor*. De igual modo, el cliente SSH y la aplicación también comparten dispositivo: *el dispositivo cliente.* Esta estrategia de comunicación proxificada a nivel local ayuda a proteger la comunicación cursada, ya que únicamente queda sin cifrar en los segmentos locales 1 y 3. Sin embargo, SSH permite que el *port-forwarding* en el segmento 3 se realice a un dispositivo diferente, tal como se aprecia en la Fig. 2.3-11 (en el caso de *port-forwarding* directo) y en la Fig. 2.3-12 (en el caso de *port-forwarding* reverso).

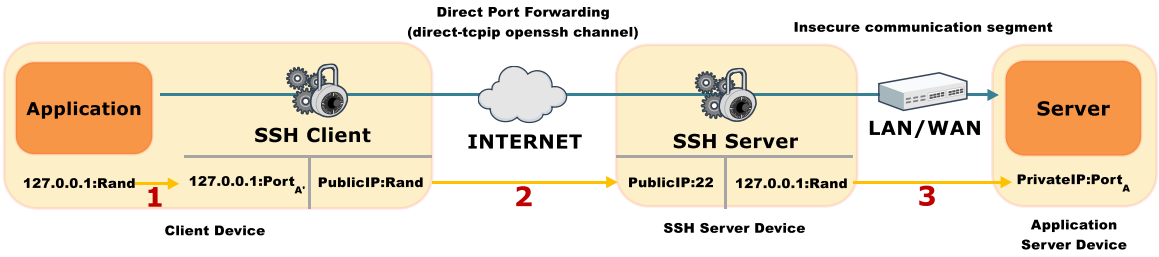


Fig. 2.3-11: Ejemplo en el que se aplica un Direct Port Forwarding para acceder a un recurso externo al SSH Server Device.

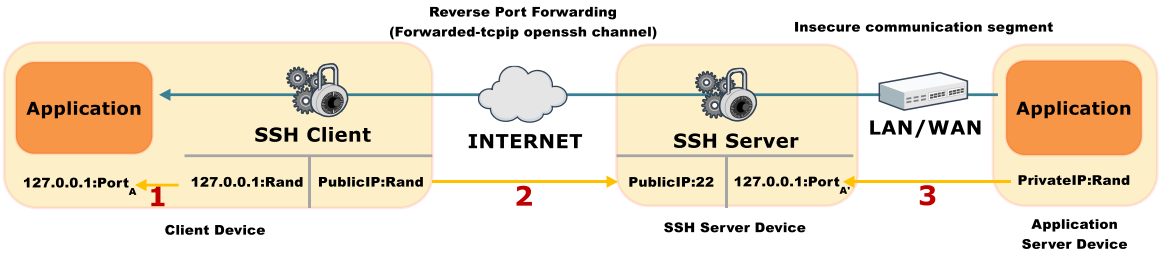


Fig. 2.3-12: Ejemplo en el que se aplica un Reverse Port Forwarding para ofrecer un recurso a un servidor externo al SSH Server Device.

De igual modo, el dispositivo cliente también podría dividirse en dos: cliente SSH y aplicación cliente, de modo que, si se llegase a combinar con los ejemplos de las Fig. 2.3-11, y Fig. 2.3-12, habría cuatro elementos en total. En este caso, los segmentos 1 y 3 seguirían estando desprotegidos por la sesión SSH y además constituirían comunicaciones no locales involucrando tráfico entre máquinas físicas distintas.

Finalmente, cabe destacar una última diferencia a nivel operativo entre los *proxy* y las proxificaciones SSH. Los *proxy* de aplicación operan normalmente a nivel de aplicación, es decir, reemplazando al destinatario de la comunicación de cara al cliente, y reemplazando al cliente de la comunicación de cara al servidor. Esto se consigue a nivel de Aplicación replicando perfectamente las primitivas de servicio ofrecidas por el protocolo de comunicación de aplicación y de seguridad empleados con el cliente y servidor respectivamente. Por el contrario, las proxificaciones SSH operan en el nivel de Transporte. Esto quiere decir que para cada tipo de servicio se requiere un *proxy* diferente, mientras que SSH puede ofrecer proxificaciones transparentes de la misma manera a cualquier servicio que utilice TCP.

En el caso particular de los *proxy* TLS, sí se observa un comportamiento parecido al de las proxificaciones SSH, al ocurrir en el nivel de Transporte. Este esquema de funcionamiento puede permitir una operación transparente a la aplicación, al igual que ocurre con la proxificación directa SSH si no se implementa autenticación TLS en el *proxy* (Fig. 2.3-13). También permite mejorar la seguridad del tráfico que opera de la zona segura hacia Internet, al poder reemplazar en tiempo real el cifrado usado en las comunicaciones que operan en Internet, siguiendo un esquema parecido al que emplearía un ataque de tipo MITM.

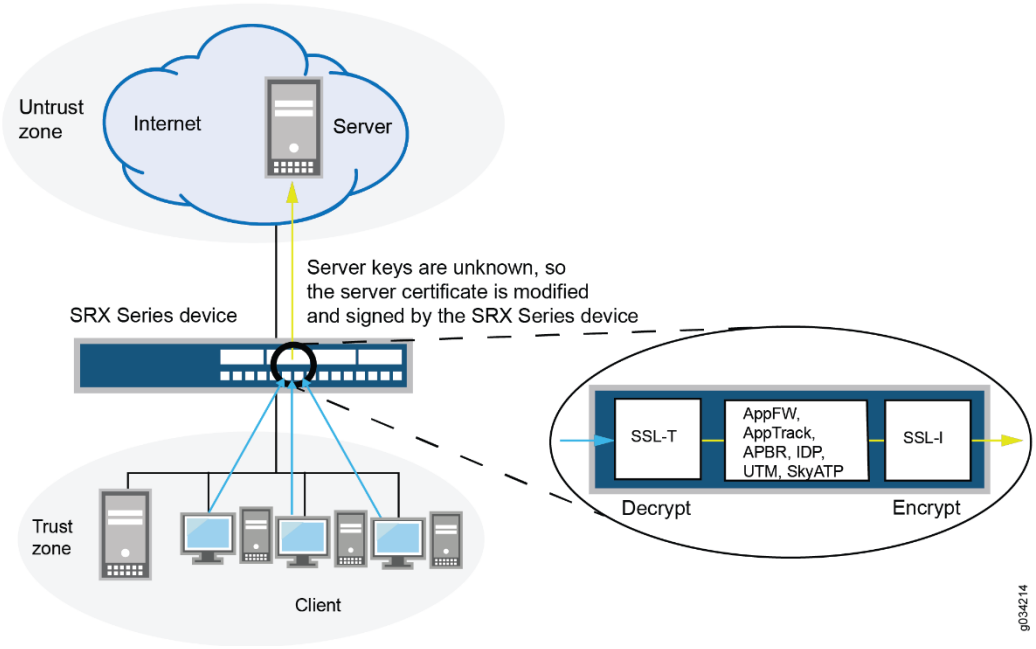


Fig. 2.3-13: Ejemplo de proxy TLS para mejorar la seguridad e la "Trust zone" al acceder a Internet[40].

En este caso particular, la principal desventaja de utilizar este procedimiento consiste en tener que confiar en todos y cada uno de los dispositivos de la zona de confianza, para que la integridad y seguridad del sistema completo se mantenga. Este tipo de procedimiento es muy similar al empleado en algunos esquemas IoT con un *Gateway,* en los que los dispositivos IoT emplean un protocolo de seguridad débil con el *Gateway,* y éste transforma la seguridad de las comunicaciones (de manera más o menos transparente) hacia el servidor. Sin embargo, existe la posibilidad de que uno de esos dispositivos pertenecientes al área de confianza se vea comprometido y sirva de plataforma para llevar a cabo un "*lateral privilege attack*" en el que "*un adversario puede comprometer un producto (dispositivo o aplicación) integrado en un hogar inteligente y escalar privilegios para relaizar operaciones protegidas en otro producto*" [86]. Un ejemplo de este problema se puede encontrar en una de las vulnerabilidades recientemente publicadas[41] que afectan a algunas bombillas inteligentes Zigbee y que permiten instalar malware en los *Gateway,* desde los que resulta factible acceder a otros dispositivos, no sólo vía Zigbee, sino mediante IP[42].

[40] Juniper Networks, "SSL Proxy", *Tech Library*, 2020, https://www.juniper.net/documentation/en_US/junos/topics/topic-map/security-ssl-proxy.html
[41] "CVE-2020-6007" *National Vulnerability Database*, 2020, https://nvd.nist.gov/vuln/detail/CVE-2020-6007
[42] "The Dark Side of Smart Lighting: Check Point Research Shows How Business and Home Networks Can Be Hacked from a Lightbulb", 2020, https://blog.checkpoint.com/2020/02/05/the-dark-side-of-smart-lighting-check-point-research-shows-how-business-and-home-networks-can-be-hacked-from-a-lightbulb/

2.3.2. SSH en dispositivos con FreeRTOS

Los microcontroladores operan con un código binario específico para su arquitectura, que se genera normalmente a partir del compilador/ensamblador que proporciona el fabricante (mediante programas o *plugin* para herramientas de desarrollo de *software*).

Al utilizar FreeRTOS para incluir un programa o aplicación en el microcontrolador, las herramientas de compilación que se utilizan normalmente generan un fichero de tipo *Executable and Linkable Format* (ELF)[43], que tiene una estructura concreta (Fig. 2.3-14) y se utiliza por el *software* específico del microcontrolador para generar el binario que se cargará en el microcontrolador. Algunos microcontroladores más avanzados presentan binarios complementarios (*bootloaders*) que configuran el arranque del dispositivo.

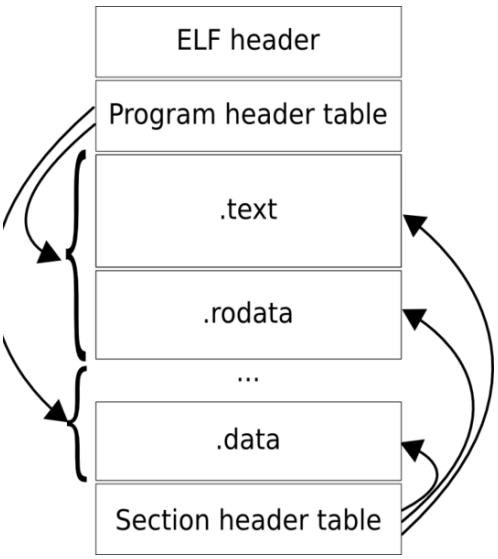


Fig. 2.3-14: Estructura de un fichero ELF: El 'program header' muestra los segmentos usados en tiempo de ejecución mientras que el 'Section header' tiene listados los grupos de secciones del binario[44].

Los sistemas operativos que se pueden cargar en este tipo de plataformas ocupan muy poco espacio y normalmente tienden a integrarse dentro del propio ELF. Esto quiere decir que todas las direcciones de memoria son ligadas estáticamente en el momento de la compilación de dicho binario y que, por norma general, en caso de necesitar modificar cualquier módulo que forme parte de él, se requiere recompilar el ELF nuevamente y regenerar el binario que se carga en el dispositivo. La forma normal de proceder con la programación de los microcontroladores que funcionan en dispositivos IoT, consiste en cargar un único binario en la ROM del dispositivo y que opera a modo de solución *bare-metal*, pero que incorpora conjuntamente la aplicación y el RTOS (Fig. 2.3-15).

[43] The Transportation and Infrastructure Service (TIS) Comitee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification", *The Linux Foundation Referenced Specification*, 1995, http://refspecs.linuxbase.org/elf/elf.pdf

[44] Loomis, J., "Executable and Linking Format (ELF)", 2008, https://johnloomis.org/microchip/pic32/elf/elf.html

Fig. 2.3-15: Estructura ROM de un microcontrolador usando FreeRTOS y su comunicación segura sobre TLS o DTLS.

Aunque los RTOS no van a ofrecer normalmente una portabilidad a nivel binaria del tipo que ofrece Linux con su ABI (para poder cargar otros ficheros binarios existentes en la ROM), sí se puede hablar de una independencia entre el RTOS y la aplicación IoT del dispositivo IoT. Este grado de independencia se traslada también a posibles implementaciones de SSH en RTOS de este tipo: La compatibilidad POSIX permite que su código fuente compile sin problemas para multitud de plataformas y se genere el ELF/BIN respectivo de cada una de ellas sin modificar el código fuente de la aplicación o del RTOS que se vaya a emplear.

2.3.3. Escalado de privilegios: Principal peligro de SSH en entornos IoT

SSH se considera un protocolo maduro que ha sido objeto de numerosos ataques a lo largo de su historia. Esto ha favorecido la aparición de vulnerabilidades que en sus sucesivas versiones se han ido corrigiendo. No obstante, versiones antiguas del mismo o implementaciones inseguras del protocolo como servicio, conducen a algunos tipos de ataques peligrosos que pueden poner en peligro, no solo el dispositivo comprometido sino infraestructuras completas (Fig. 2.3-16).

La rapidez con la que algunos dispositivos han sido puestos en el mercado impide establecer estrategias de seguridad avanzadas para servicios como SSH, distribuyéndose en ocasiones versiones obsoletas del mismo, con vulnerabilidades ya conocidas o con configuraciones estándar por defecto. Estas vulnerabilidades no suelen ser solucionadas por el usuario final, lo que facilita el éxito de ataques dirigidos contra dispositivos específicos que incorporan dicha vulnerabilidad[45], pero también de ataques masivos con el objetivo de detectar servicios SSH pobremente implementados o no mantenidos adecuadamente (Fig. 2.3-17).

[45] Kieren McCarthy, “Here's a great idea: Why don't we hardcode the same private key into all our smart home hubs?”, *The Register* 2019, https://www.theregister.co.uk/2019/07/03/zipato_hardcoded_key/

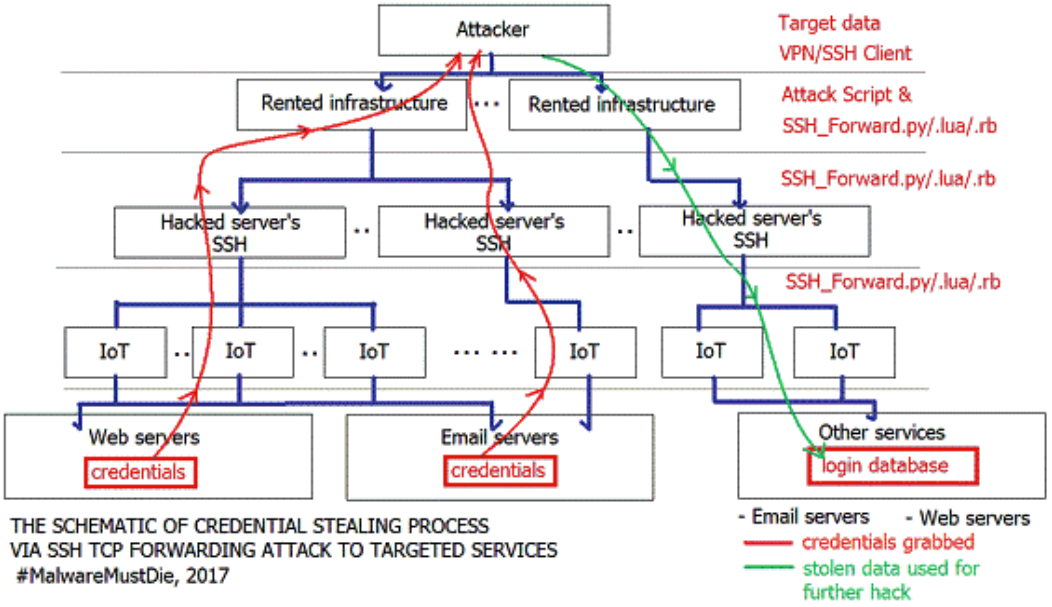


Fig. 2.3-16: SSH direct-TCPIP forward attack using weak SSH accounts[46].

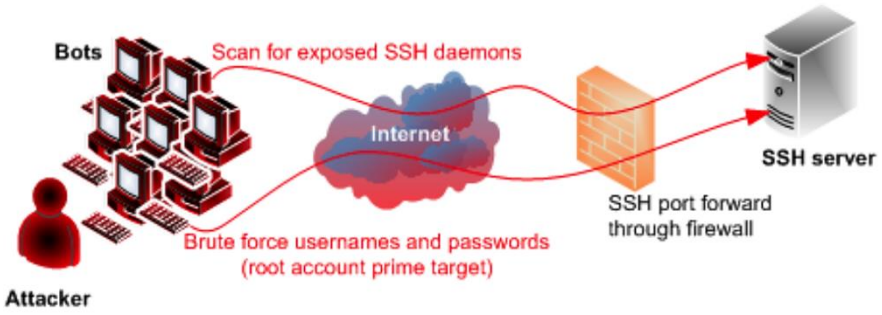


Fig. 2.3-17: "Brute force attack" usando redes de bots contra servidores SSH [87]***.***

En el caso de los dispositivos con recursos limitados (*constrained devices*), la seguridad de los protocolos tradicionales normalmente se consigue versionando esos los protocolos en implementaciones con menores requisitos, para poderlos adaptar a este tipo de aparatos. Esta adaptación a menudo se lleva a cabo de forma apresurada [2] y sin realizar todas las comprobaciones oportunas, en parte porque muchos problemas de seguridad derivados de dicha adaptación son muy difíciles de detectar y la experiencia dicta que surgirán con el tiempo, una vez se encuentre dicho protocolo funcionando. Esto explica, por ejemplo, cómo la versión de SSH para dispositivos con recursos limitados ha necesitado de numerosos parches de seguridad[47] a lo largo de 2019, de los cuales cuatro fueron orientados a solucionar vulnerabilidades calificadas como *muy importantes*.

Esto hace fundamental una actualización obligatoria y ágil de las configuraciones y de los algoritmos de seguridad cuando se consideran como potencialmente vulnerables en los

[46] Paganini, P., "A criminal group using SSH TCP direct forward attack is also targeting Italian infrastructure", *Security Affairs*, , Marzo 2017, https://securityaffairs.co/wordpress/56864/cyber-crime/ssh-tcp-direct-forward.html
[47] "WolfSSL security vulnerabilities", https://www.wolfssl.com/docs/security-vulnerabilities/

dispositivos, algo que en los aparatos IoT con recursos limitados ha sido tratado como opcional y en todo caso laborioso, pues la actualización remota y automatizada muchas veces no es posible.

2.4. DIVERSIDAD EN LA ARQUITECTURA IOT: EL PROBLEMA DE LOS “*VERTICALS*”

La competencia entre fabricantes y la ausencia de estándares de IoT han favorecido la aparición de *verticals*[48], o soluciones completas y a medida que utilizan la tecnología IoT para resolver un problema específico. Este tipo de soluciones abarcan proyectos de diversos tamaños y tipos, llegando incluso a ofrecer servicios a ciudades enteras. El problema principal de los *verticals* se caracteriza por la forma en que se aborda el problema, dándole una solución muy específica, lo que dificulta la reutilización de dicha solución en otro escenario similar. Este efecto se traduce en mayores costes de desarrollo y mantenimiento para cada proyecto, y en la necesidad de *middleware* para comunicar múltiples *verticals* a través de sus protocolos derivados. La complejidad de los sistemas aumenta de forma constante pero innecesaria, debido precisamente a la inexistencia de estándares y regulación. Esta complejidad ayuda al éxito de *hackers* (no *bots*) que realizan ataques dirigidos, pues la superposición e integración de distintas tecnologías complejas puede enmascarar errores más fácilmente. Esto encarece el mantenimiento de las soluciones, facilitando que el éxito de los ataques dirigidos pase inadvertido.

La gran cantidad de *verticals* existentes para las diferentes áreas de IoT y la necesidad de colaboración e intercomunicación entre ellas hace imprescindible la creación de supra entidades llamadas federaciones [88]. En una federación, la interconexión de diferentes *verticals* generalmente requiere *middleware* manejando “entornos semánticos transversales complejos” [89]. Por ello, el Instituto Europeo de Normas de Telecomunicaciones (ETSI), miembro de la Iniciativa Global oneM2M[49], está trabajando para ayudar en la “horizontalización” de dichos *verticals* [90], y así simplificar los procesos de intercomunicación entre ellos. Para ello, se observa la necesidad de un tipo de *middleware* horizontal estandarizado que pueda homogeneizar todos los dispositivos y tecnologías subyacentes, simplificando la intercomunicación entre servicios de diferentes “*enfoques verticales*”, como se presenta en la Fig. 2.4-1.

[48] “IoT Vertical and Topical Summit for Agriculture”, http://tuscany2018.iot.ieee.org/
[49] “Standards for M2M and the Internet of Things”, https://www.onem2m.org/

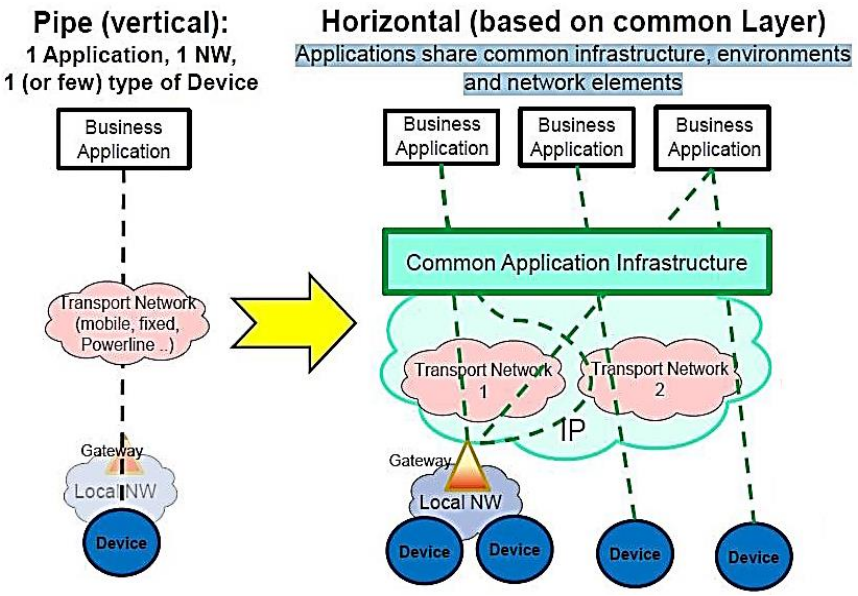


Fig. 2.4-1: Propuesta de estandarización de verticals que persiguen una horizontalización [90]**.**

La proliferación de los verticals se debe sobre todo a la innovación. La solución específica que se ofrece en cada *vertical,* independientemente de la complejidad del proyecto, puede construirse en muchas ocasiones utilizando funcionalidades sencillas que caracterizan a los dispositivos IoT. Esta versatilidad de los dispositivos, sin embargo, puede suponer un problema de seguridad de gran relevancia, debido a que el uso para el cual se diseñó el dispositivo acaba siendo diferente al que acaba desempeñado en el proyecto, y las posibles certificaciones de seguridad del dispositivo no tienen por qué seguir siendo aplicables en esos casos. "*La innovación y los costes están compitiendo con la seguridad*" [91] y por ello, la Comisión Europea y sus agencias están planteando regulaciones [13] que contemplen, entre otros, la restricción en el uso de dispositivos IoT, limitando su campo de acción explícitamente al alcance de su certificación. De ese modo, una certificación de seguridad sirve para avalar el uso concreto de un dispositivo IoT, garantizando que cuenta con las medidas de seguridad recomendadas para evitar poner en riesgo la seguridad en la infraestructura en la que se va a instalar. Además, la nueva regulación se está desarrollando con la idea de estudiar posibles corresponsabilidades en caso de un ataque, una infección o interrupción de servicio. Este hecho va a obligar a las empresas desarrolladoras a cambiar de estrategia y fabricar dispositivos que puedan ser seguros por más tiempo, y que además cuenten con métodos de actualización de seguridad remota automática (algo que también se plantea como un futuro requisito obligatorio).

2.4.1. Interconectividad a través de *Middleware* IoT seguro: El desacoplo de la seguridad

La enorme heterogeneidad de dispositivos en el mundo IoT ha fomentado la aparición de diversos *middleware* [92], que simplifican la labor de los desarrolladores de *software* IoT, al conseguir que las plataformas diseñadas ganen ubicuidad abordando múltiples tipos de

dispositivos. Estos *middleware*, obtenidos a partir de distintos *framework* de desarrollo, también han tratado de abordar el tema de la seguridad de las comunicaciones de forma general [93]. Sin embargo, una solución específica de seguridad para cada *middleware* no favorecería la interconexión entre distintos *middleware* ni el mantenimiento de dicha capa de seguridad. Esta estrategia deriva en diseños de *middleware* muy complejos que pueden acabar presentando comportamientos inesperados debido a interdependencias entre funcionalidades que no fueron contempladas y que además, resultan costosos de actualizar y mantener [94]. Por ese motivo, se ha puesto de manifiesto la necesidad de encontrar una manera estandarizada de abordar dicha seguridad que permita “*descomponer sistemas complejos en aplicaciones que comprendan componentes más sencillos y mejor definidos*” [95].

De este modo, los problemas complejos como el de la seguridad, pueden abordarse de forma más sencilla y efectiva si se tratan de forma específica e independiente cuando resulte factible. Por ejemplo, la estrategia que ha seguido la industria de pago por tarjeta (*Payment Card Industry*, PCI)[50] se basa precisamente en este tipo de desacoplo. En los últimos años, numerosos dispositivos electrónicos empleados en la industria, hostelería y comercio han incorporado el pago por tarjeta de crédito y otros medios electrónicos. Estos dispositivos, para cumplir con las normativas de seguridad vigentes, delegan la responsabilidad del pago a módulos *hardware-software* externos que han sido cuidadosamente certificados para ese fin. Esto permite libertad de diseño de los kioscos, máquinas de *vending*, cajas de supermercados, tiendas, etc. a la par que se garantiza la seguridad en las transacciones financieras. Esta idea se ha extrapolado al comercio *online*, en el que algunos establecimientos delegan el procedimiento de pago a una pasarela bancaria externa certificada. Esta estrategia de desacoplo en la seguridad del pago puede extrapolarse al resto de escenarios en los que se requiere seguridad, persiguiendo un desacoplo en la seguridad más general que proporcione simplicidad a las aplicaciones e independencia de cualquier aspecto de seguridad. De este modo, la solución de las posibles vulnerabilidades y actualizaciones de seguridad sería un problema separado del desarrollo de las aplicaciones IoT.

Siguiendo estas ideas, han surgido diversas estrategias que emplean *middleware* [96] para simplificar la labor de las aplicaciones IoT, ofreciendo normalmente una abstracción del *hardware* [92] que además incrementa la interoperabilidad y seguridad [97]. Sin embargo, estas iniciativas no pueden entenderse realmente como un desacoplo de la seguridad del *software,* tal y como ha quedado planteado en los párrafos anteriores. Aunque la aplicación nativa del dispositivo IoT disfrute de las ventajas del *middleware*, abstrayendo aspectos de *hardware* y de seguridad, esto únicamente ocurre en el diseño de la misma. Al compilarse y construirse la aplicación IoT, ocurre que el *software* continúa siendo dependiente de la versión de TLS empleada para las

[50] “Payment Card Industry certifications” https://www.pcisecuritystandards.org/

comunicaciones, y cualquier actualización del framework TLS requeriría igualmente de una recompilación de la aplicación entera.

Este problema puede apreciarse en diversas soluciones actualmente existentes: la existencia de servicios en la nube para desarrollos IoT ha propiciado la aparición de diversos *framework* que simplifican la programación de las aplicaciones y resto de *software* de los dispositivos [98]. Sin embargo, estos *framework*, aunque persigan el desacoplo de seguridad (entre otras cosas), aún continúan requiriendo del uso de unas APIs específicas (Fig. 2.4-2).

Estas APIs dependientes de los frameworks respectivos sí son susceptibles de actualizarse por temas de seguridad y de generar (aunque con menor frecuencia) inconvenientes en los programas y aplicaciones que las utilizan. Además, la portabilidad entre plataformas de distintos *framework* no es inmediata en absoluto, llegando incluso a existir problemas serios de compatibilidad entre *framework* de la misma plataforma, pero de diferente versión. Por todo ello, aún en el caso en que el *software* de los dispositivos IoT se pueda entregar en binarios diferenciados (aplicación y *middleware*), las actualizaciones de aspectos de seguridad únicamente excluirían el tener que realizar cambios en la aplicación principal del dispositivo y sí afectarían al middleware. Estos cambios, además resultarían viables únicamente cuando no afecten a la API interna que utilice la aplicación y el *middleware* para intercomunicarse. El grado de independencia del *software* nuclear de los dispositivos IoT depende por tanto en gran medida de cómo es el desacoplo de dicha seguridad en la API del *middleware* utilizado por la aplicación IoT de dicho dispositivo.

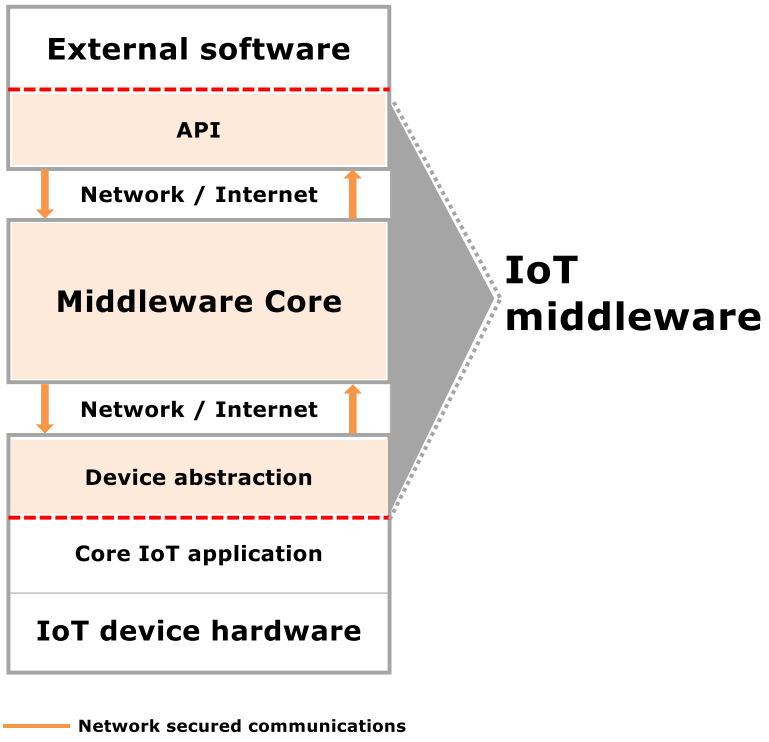


Fig. 2.4-2: Esquema estándar de middleware de seguridad usado por diversas soluciones IoT.

2.4.1.1. Ejemplos de “middlewares flexibles” que no presentan ningún avance en el desacoplo de seguridad

Entre las publicaciones existentes relativas al papel de los *middleware* en IoT, se encuentran algunos estudios [98] que valoran las posibles alternativas existentes de *framework* para IoT y de los *middleware* que estos utilizan. La mayoría de ellos, pese a la flexibilidad que puedan ofrecer a la hora del desarrollo técnológico, acaban desarrolando infraestructuras que se podrían considerar como *verticals* flexibles, pero *verticals* al fin y al cabo por el grado de dependencias *hardware* y *software* que tienen los proyectos que los utilizan (SmartThings Samsung[51], ARM Mbed[52] y Homekit Apple[53]). A nivel interno, estas soluciones se comportan de forma parecida a la *Common Application Infrastructure* definida en la Fig. 2.4-1, siempre y cuando se use su tecnología, SDKs APIs, etc. Sin embargo, los proyectos de infraestructura IoT llevados a cabo con cada *framework* de desarrollo necesitan de un *middleware* específico a nivel de *cloud* para comunicarse con otros *verticals* diferentes. Sin embargo, existen algunos *middleware* que generan menos dependencias y que se presentan a continuación [98].

Android Things

Google ha desarrollado una plataforma IoT para acelerar la implementación de las aplicaciones. Esta plataforma consiste en dos bloques: Brillo[54] y OpenWeave[55]. Brillo es un sistema operativo basado en Android para el desarrollo de dispositivos embebidos de bajo consumo energético mientras que OpenWeave ofrece la cobertura necesaria para las comunicaciones. El rol principal de OpenWeave consiste en ofrecer el soporte a los dispositivos para registrarlos en la nube y enviar y recibir los comandos remotos. Ambos componentes se complementan, y conjuntamente constituyen el *framework* IoT (Fig. 2.4-3).

El *framework* de Google se encuentra orientado principalmente al desarrollo de dispositivos IoT domésticos y se apoya en la flexibilidad de Android para abarcar un gran número de plataformas y eliminar las dependencias *hardware*. La capa HAL (*Hardware Abstraction Layer*) de Android permite homogeneizar el *hardware* subyacente evitando dependencias de las aplicaciones con el *hardware*. Sin embargo, tiene la limitación de que es necesario el uso de OpenWeave como *middleware,* y las aplicaciones desarrolladas necesitan desarrollarse empleando su *framework*, quedando incluido en su código fuente y binario. Por tanto, si aparecen actualizaciones de OpenWeave, resultará imprescindible adaptar la aplicación a ese nuevo SDK para poder recompilarla con las nuevas actualizaciones de seguridad ya incluidas en ese nuevo SDK.

[51] Samsung, “SmartThings Developer Documentation”, https://smartthings.developer.samsung.com/docs/index.html
[52] ARM, “Mbed” https://www.mbed.com/en/
[53] Apple, “The smart home just got smarter” http://www.apple.com/ios/home/
[54] Google, “Brillo”, https://developers.google.com/brillo/
[55] “OpenWeave”, https://openweave.io/

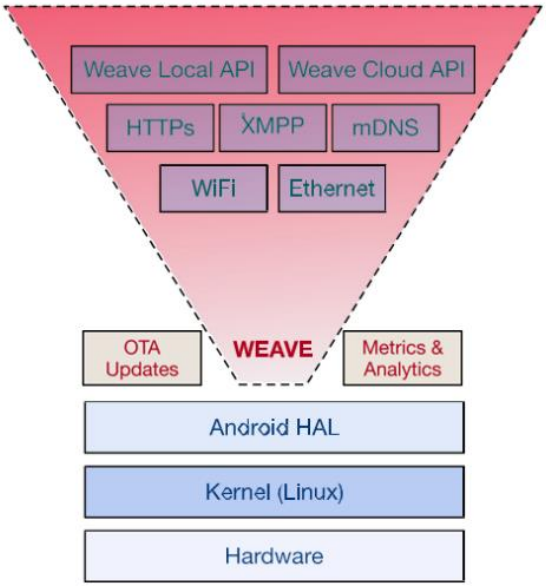


Fig. 2.4-3: Arquitectura de Brillo y de su módulo OpenWeave [97]***.***

Android Things, pretendía aprovechar la versatilidad de Android para abarcar un elevado espectro de dispositivos y soluciones (independencia de *hardware*). Sin embargo, ha acabado convirtiéndose en un *vertical,* orientándose casi únicamente a dispositivos con Google Assistant incorporado, que se distribuya integrado en los dispositivos desde fábrica siguiendo una política OEM[56].

Amazon Web Services (AWS) IoT

La plataforma IoT que ofrece Amazon[57] dispone de un *framework* que permite conectar fácilmente los dispositivos IoT a la nube. Además, este *framework* simplifica la interacción de esos dispositivos con otros servicios AWS, como Amazon DynamoDB[58] (Base de datos NoSQL), Amazon S3 (Amazon *Simple Storage Service*)[59] o Amazon *Machine Learning*[60] [98].

Los dispositivos IoT que requieran implementar sus aplicaciones en AWS deben utilizar la SDK de desarrollo para los dispositivos. Esta SDK integra toda la lógica de seguridad empleada por Amazon AWS que permite comunicar los dispositivos con los servicios de la nube de forma ágil (Fig. 2.4-4). Cualquier proyecto que incorpore la tecnología AWS IoT puede interconectarse fácilmente con otros proyectos que compartan su misma tecnología. Sin embargo, la SDK impide que la seguridad pueda tratarse de forma independiente a la aplicación, causando además que la conectividad con sistemas no AWS requiera nuevamente de *middleware* intermedios en la nube. Por otro lado, los problemas relativos a la actualización de seguridad estudiados

[56] Dave Smidth, "An Update on Android Things", *Android Developer's Blog*, 2019, https://android-developers.googleblog.com/2019/02/an-update-on-android-things.html
[57] "AWS IoT *framework*". https://aws.amazon.com/iot
[58] "DynamoDB". https://aws.amazon.com/dynamodb
[59] "Amazon s3". https://aws.amazon.com/s3
[60] "Amazon machine learning". https://aws.amazon.com/ machine-learning

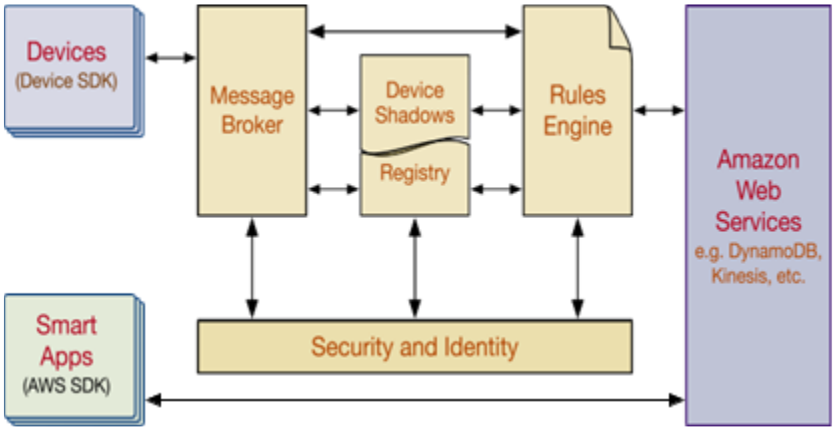


Fig. 2.4-4: ***Esquema de seguridad seguido por AWS IoT*** [98]**.**

anteriormente se hacen patentes de nuevo en este esquema de comunicación, al no existir un desacoplo explícito entre las aplicaciones y su esquema de seguridad, muy interrelacionado con la SDK de los dispositivos. De este modo, si se detecta algún tipo de vulnerabilidad que requiera modificar la SDK utilizada en el desarrollo de las aplicaciones de los dispositivos IoT, al adaptarse a la nueva SDK, podrían tener que editarse partes sensibles del código fuente de la aplicación, siguiendo las guías de migración pertinentes, para poder recompilarse correctamente[61].

Microsoft Azure

Microsoft Azure es una plataforma que proporciona diferentes tipos de funcionalidades orientadas a proyectos IoT. Este conjunto de herramientas ofrece al desarrollador la facilidad de interactuar con los dispositivos IoT y trabajar con los datos que estos generan, para poder presentarlos como servicios de los que se deriven oportunidades de negocio (Fig. 2.4-5).

Microsoft proporciona varias SDK para ofrecer soporte a distintos tipos de plataformas y dispositivos IoT de diferentes arquitecturas. Además, permite a los desarrolladores trabajar con diferentes lenguajes de programación. Estas SDK permiten a su vez integrarse de forma muy sencilla con los servicios ofrecidos nativamente por múltiples sistemas operativos[62].

Las SDKs de los dispositivos IoT incluyen TLS como protocolo de comunicación. Esto quiere decir que las aplicaciones desarrolladas para estos dispositivos, no sólo encuentran dependencia con la arquitectura IoT de Azure, sino también con la seguridad que ofrece. Por tanto, si es necesario realizar actualizaciones de las SDK, las aplicaciones deberán ser adaptadas a la nueva SDK, recompiladas y actualizadas.

[61] “AWS IoT Device SDK C, Migration guide”, *AWS Documentation*, https://docs.amazonaws.cn/en_us/freertos/latest/lib-ref/c-sdk/main/migration.html
[62] “Catálogo de dispositivos Azure certificados para IoT”, https://catalog.azureiotsolutions.com/alldevices

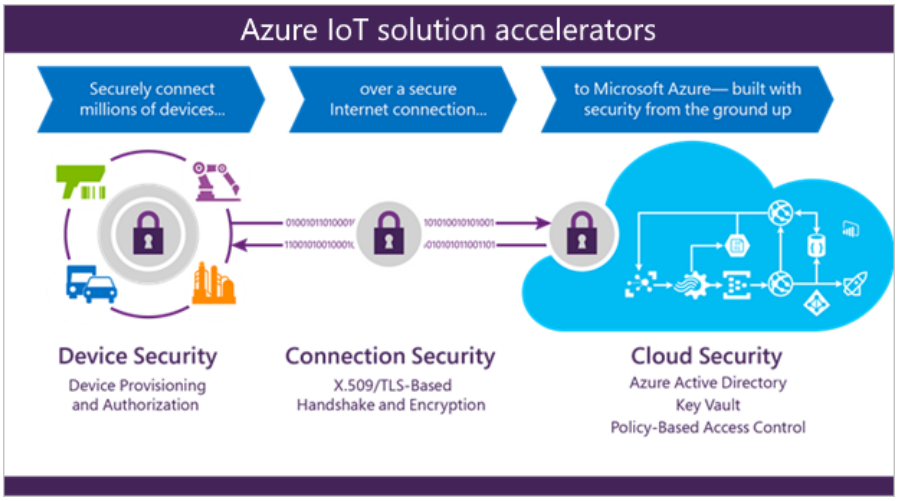


Fig. 2.4-5: Arquitectura de seguridad de Azure[63].

No obstante, el equipo de desarrollo de Microsoft Azure, consciente del reto que esto representa en muchos casos, ofrece versiones de SDK con soporte a largo plazo (*Long Term Support*, LTS). Esto permite mantener operativos por más tiempo aspectos críticos de las SDKs, para evitar actualizaciones de código fuente ineludibles, que obliguen a tener que rediseñar las aplicaciones IoT por cambios no deseados en la SDK[64]. El enorme esfuerzo que esto representa para Microsoft justifica que, pese a ofrecer soporte en una gran cantidad de dispositivos y sistemas operativos, únicamente una pequeña parte de ellos se encuentre certificada por Microsoft Azure.

La plataforma IoT presentada por Microsoft ofrece un amplio abanico de posibilidades para adaptarse a multitud de entornos y arquitecturas. La flexibilidad y el soporte de seguridad se consiguen con múltiples variedades de SDKs para distintas tecnologías y lenguajes que Microsoft debe mantener simultáneamente. Microsoft Azure combate la verticalidad inherente a trabajar con sus *framework* de desarrollo, generando múltiples SDKs de arquitecturas específicas. Estas sirven a su vez para desarrollar de forma sencilla *middleware* complementario, que sirve para facilitar la interoperabilidad entre arquitecturas y sistemas, acercándose de forma sustancial al objetivo planteado por la ETSI: diseñar una infraestructura de aplicación común que permitiese "horizontalizar los *verticals*" (Fig. 2.4-1). El principal inconveniente de estas soluciones basadas en Microsoft Azure es la complejidad existente en ecosistemas IoT no homogéneos. A cambio de la interconectividad se requiere el uso de una amplia diversidad de SDKs y *middleware* diferentes, lo que eleva su coste y complica el mantenimiento.

[63] "Seguridad del Internet de las Cosas (IoT) desde el principio", *Microsoft Docs* , https://docs.microsoft.com/es-es/azure/iot-fundamentals/iot-security-ground-up

[64] "SDKs for a variety of languages and platforms that help connect devices to Microsoft Azure IoT services", https://github.com/Azure/azure-iot-sdks

RERUM

Este proyecto[65] se llevó a cabo a través de un consorcio financiado por la Unión Europea para el desarrollo de tecnología aplicable a las ciudades inteligentes (*smart cities*). Su objetivo consistió en desarrollar, evaluar y poner a prueba una arquitectura IoT y su *framework* correspondiente, que permitiese dar soporte de forma confiable y segura a dispositivos inteligentes de características heterogéneas utilizados en las ciudades. El principal objetivo de este *framework* consistió en abordar estos objetivos, considerando en todo momento el concepto de “seguridad y privacidad desde las fases de diseño” (“*security and privacy by design*”).

El *framework* diseñado trató de dar cobertura a la mayor cantidad de dispositivos posibles, para conformar un *middleware* de interconectividad seguro y confiable [99]. Este objetivo puede entenderse como una manera de fomentar interconexión entre *verticals,* gracias a este *framework* tan flexible para soluciones IoT de múltiples tipos. Esta flexibilidad la consigue gracias a una abstracción de los dispositivos de cara al *middleware* central (Fig. 2.4-6). Esta abstracción se lleva a cabo en los dispositivos IoT, convirtiendo los aspectos *hardware* de dicho dispositivo en un “dispositivo RERUM” (*Rerum Device,* RD). Esta entidad virtual es más sencilla de manejar, al resultar homogénea independientemente del dispositivo IoT abordado, lo que permite que los módulos del *middleware* principal en los servidores se beneficien de esa abstracción de todos los recursos y servicios de los dispositivos IoT, en forma de recursos de dispositivos RERUM.

Esta estrategia de “virtualización” trata de acercar a RERUM hacia ese *middleware* conceptual propuesto por la ETSI, que proporcione una Infraestructura de Aplicación Común (*Common Application Infrastructure*) en la Fig. 2.4-1. Sin embargo, esta abstracción no puede entenderse como un desacoplo de seguridad totalmente efectivo, ya que los módulos RD de los dispositivos IoT y las funcionalidades de seguridad no son independientes. El adaptador RD forma parte del entorno del *middleware* RD que se incluye en cada dispositivo IoT para homogeneizarlo, pero este adaptador también incluye las funcionalidades responsables de la seguridad en las comunicaciones.

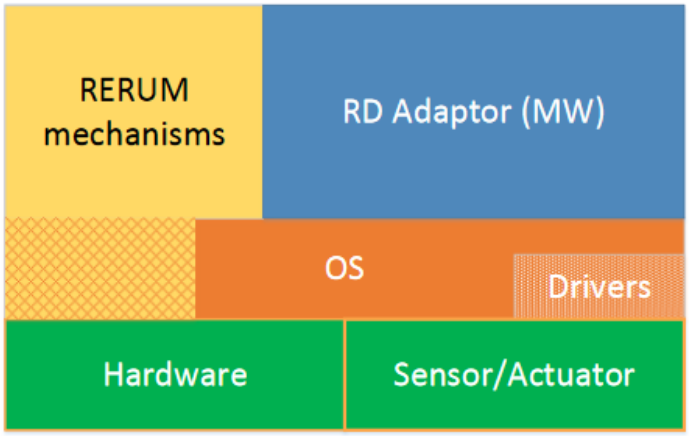


Fig. 2.4-6: Esquema RERUM device functional layers [99]**.**

[65] “RERUM: REliable, Resilient and secUre IoT for sMart city applications”, https://ict-rerum.eu/

Además, existen otros problemas para poder entender a RERUM como realmente una Infraestructura de Aplicación Común. No resulta trivial incorporar dispositivos de tecnologías diferentes y ya desplegadas al sistema RERUM, ya que requeriría de cambios sustanciales en el diseño del *software* de los dispositivos IoT. Por otro lado, esta técnica de abstracción de *hardware* se encuentra limitada únicamente a arquitecturas ARM [100].

2.4.1.2. Middleware que presentan avances en el desacoplo de seguridad

La preocupación por la seguridad, la necesidad de interoperabilidad entre verticals y los esfuerzos regulatorios actuales representan intereses contrapuestos de actores distintos que, al estar interrelacionados, acaban frenando la mayoría de iniciativas que permitan avanzar. Este hecho ha acabado motivando un tratamiento diferenciado de la seguridad, de manera que sea manejada aparte de otras funcionalidades tradicionales que el dispositivo IoT pueda ofrecer.

Los procesadores y microcontroladores modernos proporcionan funcionalidades especiales que pueden ayudar a lograr este objetivo. Las tareas relacionadas con el manejo de la seguridad, la integridad de las comunicaciones y la confianza entre las aplicaciones de los dispositivos se pueden abordar con elementos *hardware* específicos que conforman un entorno de ejecución seguro o *Trusted Execution Envirnment* (TEE) [101]. Esto puede entenderse como una especie de desacoplo de seguridad de las aplicaciones en los dispositivos, de manera que las aplicaciones puedan delegar las principales responsabilidades en el campo de la seguridad sobre estas entidades, generando además confianza. Las aplicaciones que se ejecutan bajo este paradigma se consideran bajo un TEE [102]. A continuación, se presentan algunos de los framework que tratan de desarrollar esta estrategia de seguridad.

GlobalPlatform

Las aplicaciones que se pretenden desarrollar empleando TEE, deben emplear un *framework* TEE de desarrollo diferente según la arquitectura elegida, ya que el desacoplo de seguridad hacia elementos *hardware* es muy dependiente de la tecnología de seguridad hardware (ARM TrustZone[66], Intel Software Guard Extensions[67], RISC-V Multizone[68], etc.). Sin embargo, este tipo de esquemas convierten el desarrollo de seguridad de *software* en un producto muy dependiente de la arquitectura *hardware* dedicada que se pretenda utilizar. Por ello, han surgido iniciativas como Globalplatform[69] que congrega a más de 90 organizaciones de todo el mundo con el objetivo de diseñar las bases de un conjunto de estándares que puedan

[66] "Arm TrustZone Technology", *Arm Developer*, https://developer.arm.com/ip-products/security-ip/trustzone
[67] "Intel Software Guard Extensions", *Software Intel*, https://software.intel.com/es-es/sgx
[68] "MultiZone Security for RISC-V", *Hex Five Security*, https://hex-five.com/multizone-security-sdk/
[69] "Global Platform, The standard for secure digital services and devices", https://globalplatform.org/

emplear cualquier tipo de arquitectura con *hardware* de seguridad dedicado. De este modo, se persigue también ganar en interoperabilidad, ya que los diseños *software* bajo este nuevo esquema son fácilmente recompilables en diferentes arquitecturas *hardware*. Entre los estándares de seguridad definidos por GlobalPlatform, existe uno dedicado explícitamente a las comunicaciones y que toma forma de API: el TEE *Sockets* API [103].

Esta API habilita a las aplicaciones de confianza (Trusted Applications, TAs) para que puedan establecer y emplear sus comunicaciones de red a través de un nuevo tipo de *socket* de red denominado *iSocket*. Esta entidad actúa como empaquetador (*wrapper*) de un *socket* de red estándar para la TA (Fig. 2.4-7). Las aplicaciones del dispositivo IoT que requieren las funcionalidades ofrecidas por las TA (ejecutadas en el Entorno Enriquecido de Ejecución, REE según GlobalPlatform) deben utilizar una API de cliente TEE, para poder así dotar de seguridad a sus comunicaciones a través de la TA. Esta estrategia sí representa en primera instancia un desacoplo de seguridad más completo de la aplicación IoT. El *iSocket* puede ser configurado para establecer una comunicación segura y ésta se establecerá por la TA utilizando TLS o DTLS. En el caso en que esos protocolos queden desactualizados, se puede diseñar una nueva TA que actualice a la existente, para emplear así la nueva versión de TLS en las comunicaciones. Implementar este tipo de actualización a la TA en los dispositivos IoT no tiene por qué ser sencillo, pero en todo caso supone un procedimiento mucho más simple que actualizar la aplicación REE completa para cada tipo de dispositivo.

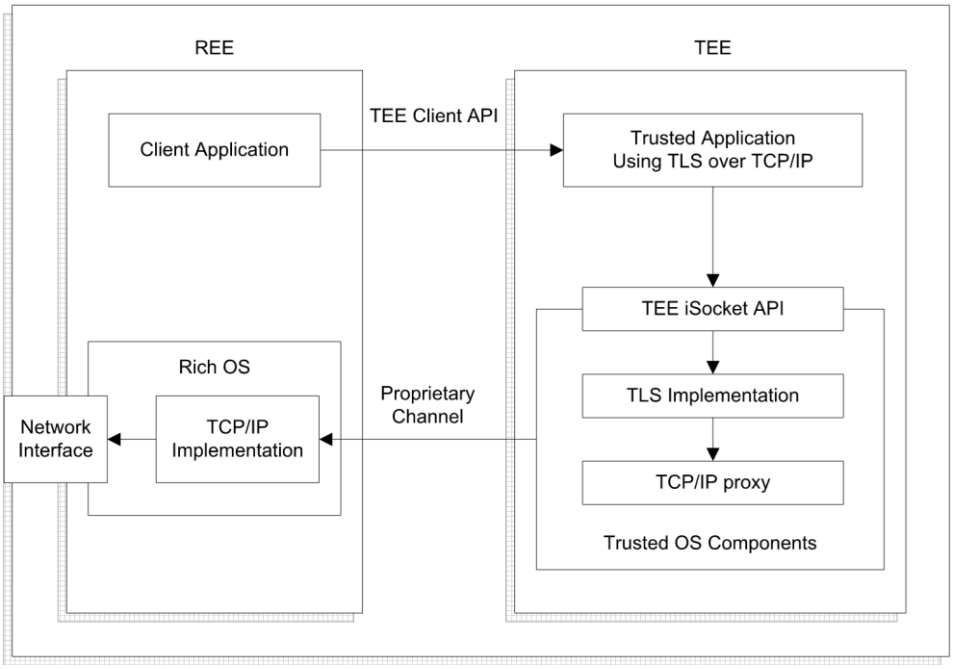


Fig. 2.4-7: Separation of Security Protocols and Pure Transport Protocols en GlobalPlatform [103]**.**

Este elemento en particular (la TA) además se presupone común en múltiples dispositivos y bajo desarrollo y soporte constante, y en todo caso independiente de las aplicaciones REE que la acaben utilizando. Sin embargo, si la actualización (de la TA) requiere también una actualización de la API del cliente TEE, la aplicación REE tendrá que adaptarse también y su código binario deberá ser reemplazado. En todo caso, esta situación únicamente se plantearía para actualizaciones significativas que requieran de cambios de APIs, como los vistos en otros *middleware* de carácter vertical pero que, al tener que producirse tarde o temprano, acaban limitando en el tiempo el desacoplo efectivo de la aplicación con la seguridad.

Open-TEE

Open-TEE [104] es una tecnología diseñada inicialmente para ayudar a los desarrolladores de aplicaciones bajo entornos TEE que cumplen con las especificaciones de GlobalPlatform. Para ello, Open-TEE ofrece un espacio virtualizado en el que las funcionalidades de seguridad diseñadas para el *hardware* específico TEE operan a nivel de software. Esta característica permite simular por *software* las funcionalidades *hardware* TEE en arquitecturas en las que no están disponibles. Por ello, esta herramienta ha empezado a considerarse también como una solución TEE a la que recurrir por defecto en ausencia de *hardware* específico. Esta solución permite a una aplicación diseñada para ejecutarse en entornos TEE operar de forma análoga en una arquitectura *hardware* sin ese soporte específico. Esta abstracción del *hardware* no resulta tan efectiva ni eficiente como el empledo de *hardware* de seguridad dedicado. Sin embargo, constituye un gran avance hacia el desacoplo de la seguridad, aunque presenta un problema importante también detectado en las soluciones GlobalPlatform estándar: las comunicaciones con dispositicos no TEE. En el caso en que una aplicación TEE requiera comunicarse con sistemas que no soportan los esquemas de seguridad basados en TEE, la aplicación debe programarse hacer uso ella misma de los protocolos de comunicación segura (TLS/DTLS), lo que se traduce en una pérdida del desacoplo de seguridad.

Desacoplo de seguridad por virtualización

Otras alternativas persiguen ahondar más en la virtualización como medida de seguridad [97]. Esta estrategia sirve para aislar aplicaciones entre sí, y de ese modo conseguir una cadena de confianza más robusta a la hora de ejecutar aplicaciones arbitrarias. Para ello, basta con que la arquitectura *hardware* a emplear cuente con prestaciones *hardware* de virtualización (Fig. 2.4-8). Este tipo de virtualización se ha conseguido hacer funcionar en dispositivos muy limitados (por ejemplo, con 2 Mbytes de memora ROM y 512 Kbytes de RAM [97]). Sin embargo, la protección en las comunicaciones existente entre máquinas virtuales no es extrapolable a las comunicaciones que las aplicaciones de las máquinas virtuales tienen con el exterior, por lo que la seguridad de estas últimas comunicaciones debe correr nuevamente a cargo de las aplicaciones ejecutadas en las máquinas virtuales y, por tanto, estas aplicaciones deberán incorporar los protocolos de seguridad pertinentes.

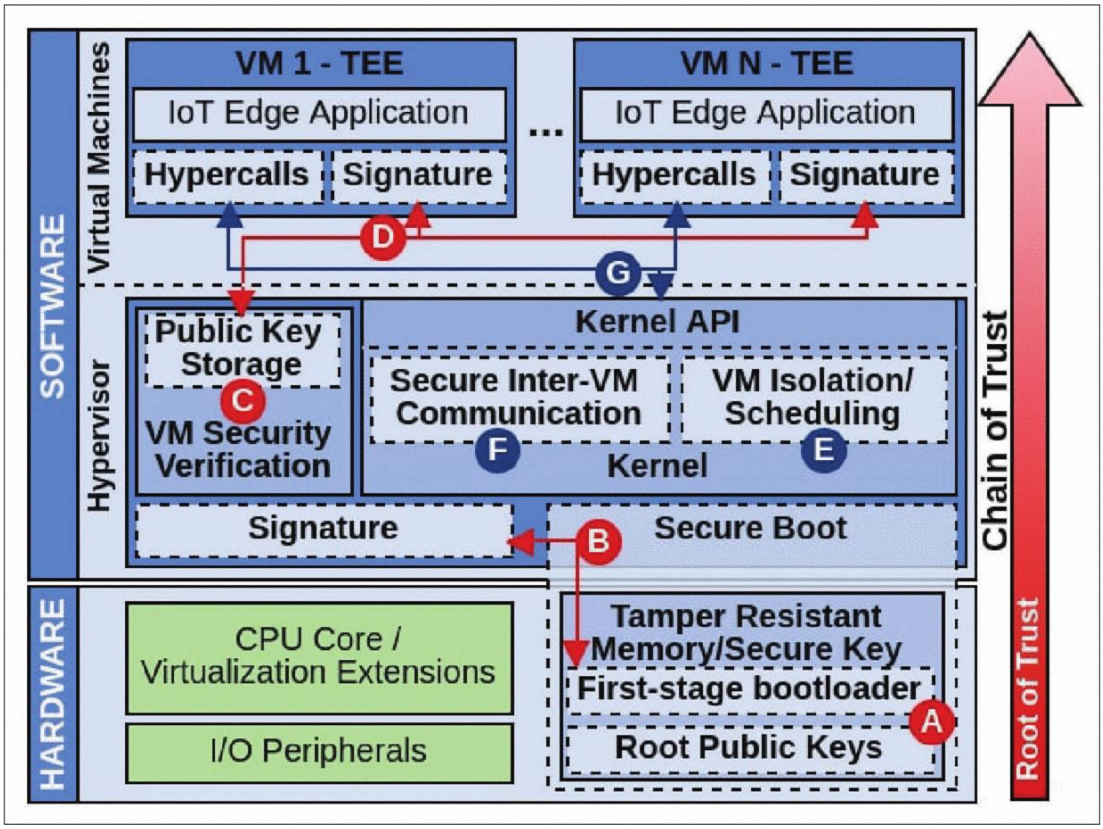


Fig. 2.4-8: Cadena de confianza preservada gracias a la virtualización embebida [97]**.**

SμV: The Security MicroVisor

Existen dispositivos IoT todavía más restringidos, como los de clase C1 [70] (10 Kbytes de RAM y 100 Kbytes de ROM), que no cuentan con *hardware* dedicado (ni de seguridad ni de virtualización), para los que puede resultar interesante el contar con medidas de ejecución en entornos TEE [105]. En estos casos en los que tampoco se cuenta con Unidad de Gestión de Memoria (*Management Memory Unit*, MMU), se pueden incorporar mínimas medidas de seguridad para el aislamiento y control en la ejecución del código con estrategias de protección basadas en un micro *hypervisor* que proteja al dispositivo de aplicaciones inseguras que traten de alterar la propia aplicación o el mismísimo *hypervisor*. También se protege el dispositivo de ataques externos que traten de desencadenar un desbordamiento de *buffer* para conducir a una ejecución de código arbitrario [106] (Fig. 2.4-9).

En este tipo de dispositivos, la seguridad en las comunicaciones se lleva a cabo de forma integrada en la propia aplicación que constituye su *firmware*. Este hecho de por sí y de forma general dificulta avanzar en el desacoplo de la seguridad de las comunicaciones. Para proceder con las actualizaciones de forma remota, además se requerirían funcionalidades complementarias para ello residentes en el *hypervisor*.

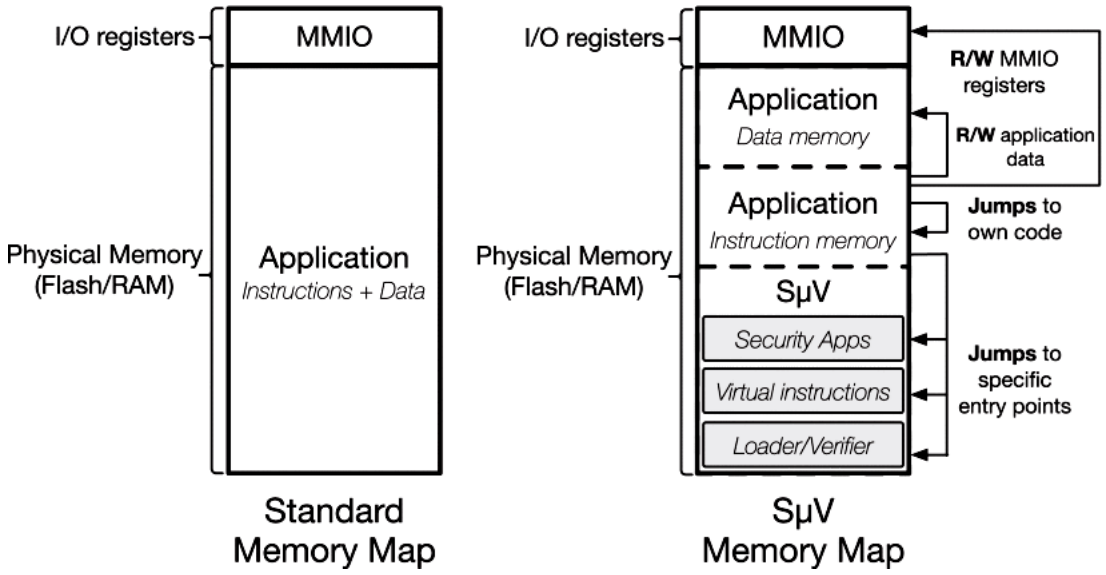


Fig. 2.4-9: ***Mapeo de memoria estándar en dispositivos C1 sin MMU (izquierda) y mapeo de memoria con el micro hypervisor SμV. En el caso estándar, la memoria es monolítica y no existen restricciones de operación. En el caso gestionado por SμV, la memoria queda dividida en el campo de instrucciones y el de datos y se restringe la acción de operaciones de lectura/escritura/ejecución consideradas como* sensibles [106].

3. PRECURSORES, DESCRIPCIÓN, DESARROLLO Y ANÁLISIS DE LA SOLUCIÓN PROPUESTA

Como ya se ha dicho, esta tesis analiza la problemática actual de las telecomunicaciones en entornos IoT. Este tipo de problema se está tratando de abordar de distintas maneras por medio de *middeware* de seguridad [96], ofreciendo entre otros, abstracción de *hardware* [92] que trate de incrementar la interopreabilidad y la seguridad [97]. Sin embargo, tal y como se ha visto en el capítulo anterior, las APIs requeridas por las aplicaciones y el *software* del dispositivo IoT, impuestas por el *framework* de desarrollo empleado, pueden acabar impidiendo un verdadero desacoplo de las funciones de seguridad.

Una posible solución que termine con el problema de la interdependencia entre las funcionalidades de seguridad y las propias del dispositivo IoT ofrecidas por sus aplicaciones, consiste en aprovechar una interfaz imprescindible en todos los esquemas IoT de comunicación: los *sockets* de red (API de *socket*) que, junto con las funcionalidades el sistema operativo de tipo POSIX o RTOS, comienzan a ser omnipresentes en los diseños. Si las APIs de seguridad de los *middleware* se pudieran sustituir por una solución transparente al *software* IoT que empleara únicamente estos dos elementos, el desacoplo de seguridad sería totalmente efectivo [107] [108].

En este capítulo se analizan algunos proyectos precursores de la solución, en los que trabajó el autor de esta Tesis. En particular, a lo largo de la sección 3.1 se presentan diversos proyectos y pruebas de concepto que se elaboraron, y que ponen de manifiesto algunas de las características de los dipositivos IoT descritas anteriormente. Además, se incluyen las estrategias de desarrollo que condujeron a la búsqueda de un esquema que facilitara el desacoplo de seguridad. Pensamos que puede resultar de interés conocer las fases previas de trabajo, que condujeron a la solución finalmente adoptada, que denominamos IoTsafe.

En la sección 3.2 se presenta IoTsafe como la solución propuesta y se analizan los conceptos básicos que la caracterizan. En la sección 3.3 se incluye una descripción detallada de la solución que permite definir mejor el comportamiento de una implementación completa. En la sección 3.4 se analiza IoTsafe desde una perspectiva de la ITU para tratar de definirlo como un *framework* de transporte desde el punto de vista de las recomendaciones ITU. Finalmente, la sección 3.5 ofrece un análisis de las principales ventajas de la solución.

3.1. PRECURSORES EN LA BÚSQUEDA DE UN NUEVO SISTEMA DE COMUNICACIÓN SEGURO

Como esquema de comunicación segura, IoTsafe surge a partir de la necesidad de encontrar una manera más efectiva de comunicación para dispositivos *Digital Signage* (DS) [109] ya considerados parte de la IoT. Los dispositivos DS suelen consistir en pantallas audiovisuales gestionadas de forma autónoma que presentan información de interés. Estos dispositivos pueden

ser interactivos, por medio de la pantalla, de otros sensores u otros sistemas de información (Fig. 3.1-1). Esta tecnología persigue principalmente automatizar algunos procesos relativos a la comunicación audiovisual, aplicable en mercadotecnia, comunicación interna empresarial, comunicación institucional o en servicios complementarios de otros sistemas para *Smart Cities*, etc.

El *software* de estos dispositivos puede utilizar el patrón de arquitectura Modelo Vista Controlador (MVC). La filosofía que soporta este tipo de desarrollo de *software* busca un grado de desacoplo que no sólo afecte a los distintos elementos de la arquitectura, sino a las capas de comunicación empleadas. La necesidad de IoTsafe surgió durante el desarrollo de proyectos IoT relacionados con DS, en los que trabajó el autor de esta Tesis. En este campo, resultó fundamental el poder reutilizar *software* y aplicaciones, al mismo tiempo que se simplificase la interoperabilidad entre los distintos módulos, para poder ofrecer soluciones a medida más rápidamente, independientemente de la tecnología utilizada.

Fig. 3.1-1: Ejemplos de aplicación de la Digital Signage de Servicios de TI de Durango SA de CV.

La arquitectura de las aplicaciones DS presentadas a continuación sigue un esquema MVC clásico. Tradicionalmente, algunas soluciones basadas en MVC incluyen los tres elementos descritos en la Fig. 3.1-2 en un mismo programa.

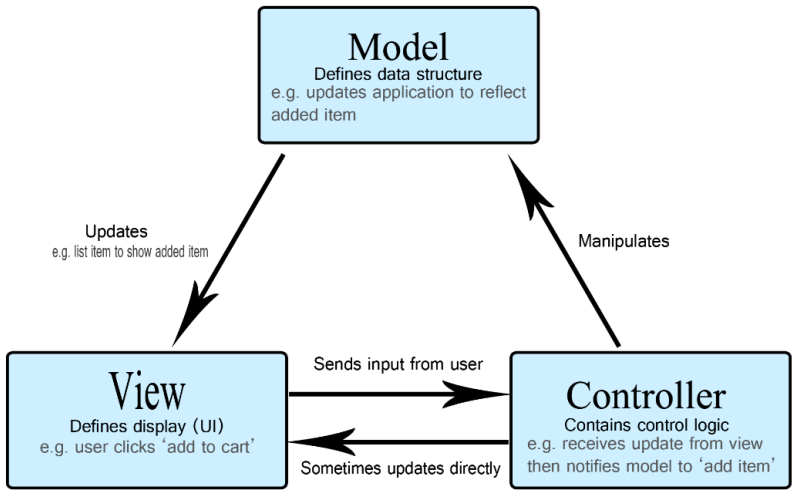


Fig. 3.1-2: Relación entre los distintos elementos del patrón de arquitectura MVC[70].

La función del Modelo, es llevada a cabo normalmente por una base de datos, que puede desarrollarse de forma independientemente al programa principal y comunicarse con ella opcionalmente a través de conexiones ordinarias TCP (Fig. 3.1-3). Además, en este tipo de desarrollos orientados particularmente a entornos web, el Controlador y la Vista son accedidos remotamente por exploradores web y, por tanto, ambos elementos tampoco tienen por qué encontrarse siempre integrados en una misma aplicación.

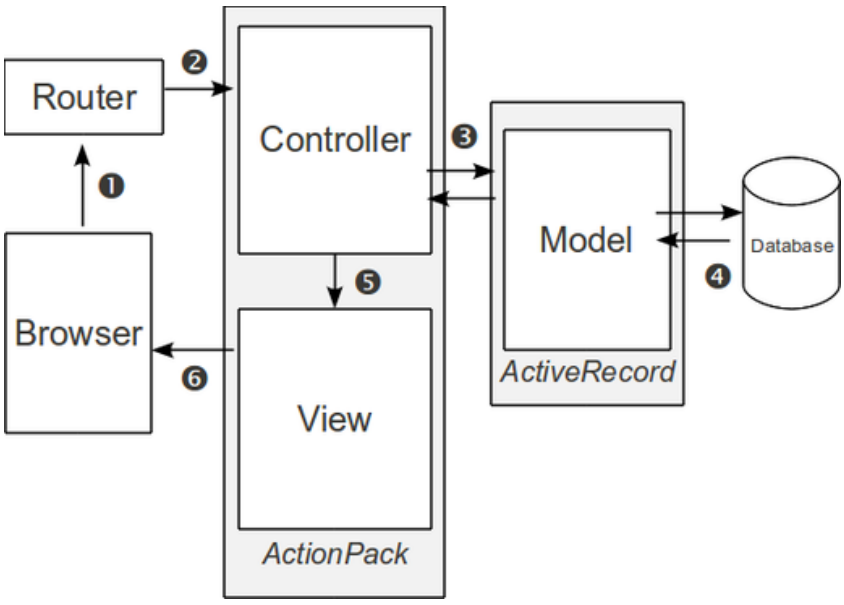


Fig. 3.1-3: Diagrama de interacciones típico de aplicaciones web basados en Ruby on Rails[71].

[70] "Glosario: MVC", *MDN Web docs*, 2020, https://developer.mozilla.org/es/docs/Glossary/MVC
[71] Introduction to Ruby on Rails, MVC Architecture, http://www.cs.mun.ca/~donald/courses/2011-Winter/comp3710/rails/

En los desarrollos DS en los que se trabajó, se valoró la idea de desacoplar totalmente la vista del controlador de forma equivalente a como opera el sistema de ventanas X de Unix/Linux (Fig. 3.1-4). La principal diferencia consistió en sustituir el protocolo X de comunicación interna por HTTP. De este modo, toda la información audiovisual mostrada en pantalla se transmite del controlador a la vista de forma local y vía HTTP (sin seguridad). Este tipo de diseño, permite a su vez que los gestores del controlador puedan obtener fácilmente vistas remotas al emplear también HTTPS (con seguridad) y utilizar para ello algún tipo de *proxy* operando como *middleware* que convierta HTTP a HTTPS.

La idea de utilizar HTTP como protocolo de comunicación con la vista, es común en otro tipo de diseños MVC constituidos en al menos dos dispositivos diferentes (Fig. 3.1-5). El uso de este protocolo para la comunicación entre la vista y el resto de elementos mejora notablemente la independencia y permite que, una vez se establezcan las condiciones de seguridad adecuadas, se pueda establecer una vista remota de la aplicación más fácilmente.

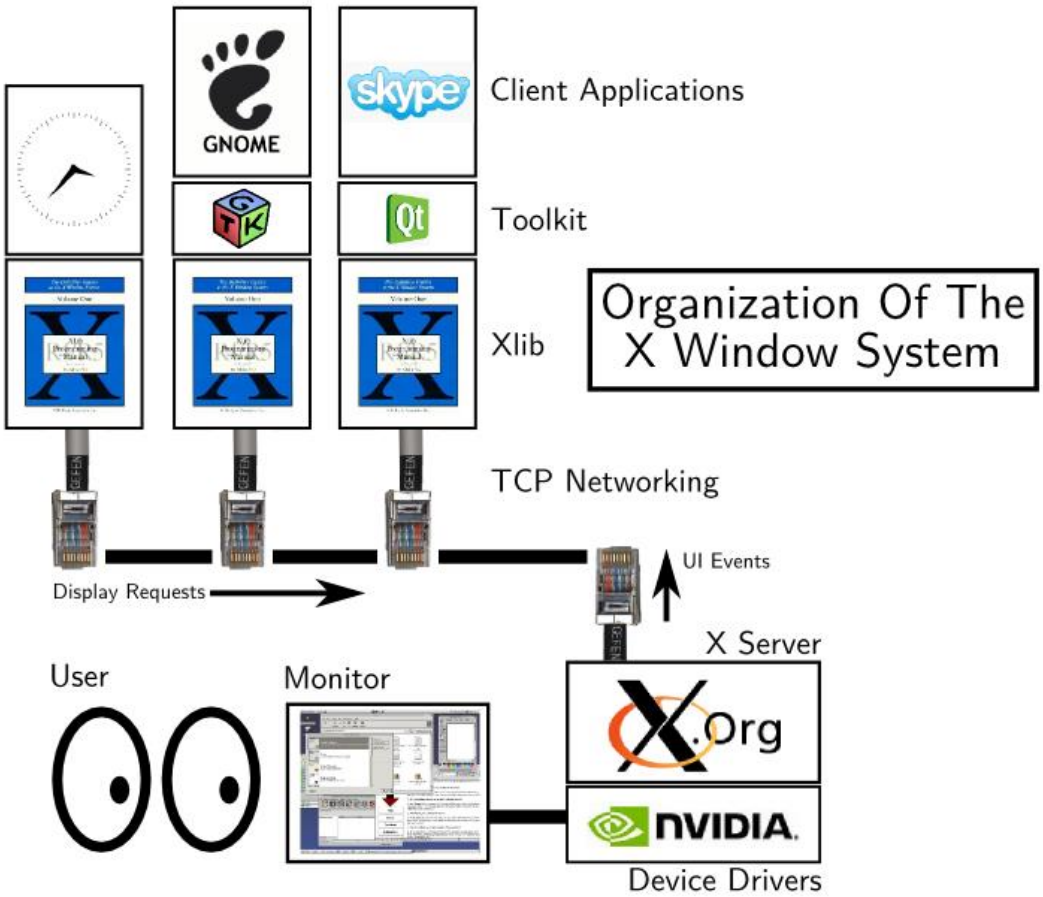


Fig. 3.1-4: Estructura del sistema de ventanas X Windows[72].

[72] Chris X Edwards, "Using The X Window System Remotely", 1998, http://xed.ch/h/remoteX.html

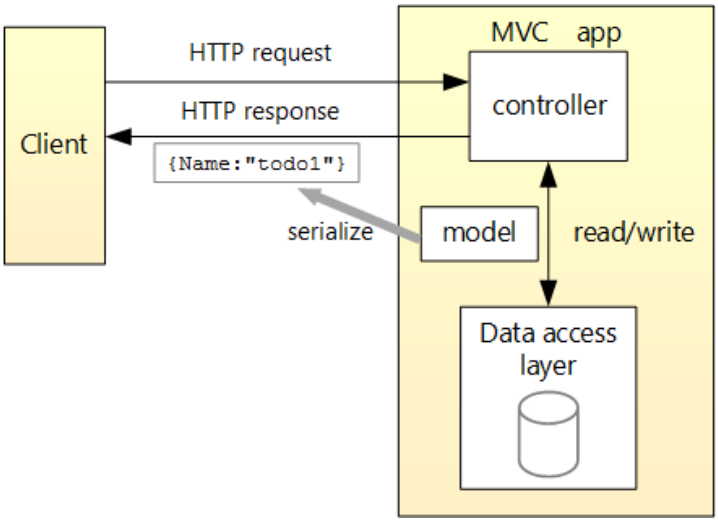


Fig. 3.1-5: Diagrama de aplicación estándar en ASP.NET, en el que la vista usada por el cliente se encuentra más claramente desacoplada del controlador [73].

Sin embargo, para evitar la complejidad creciente en el controlador (por el *middleware* asociado a la gestión remota y a la seguridad), se acabó optando por investigar nuevas estrategias de comunicación que permitiesen en el futuro ofrecer los niveles de seguridad requeridos sin necesidad de sobrecargar el controlador con esas funciones. Además, dado que este desarrollo se consideró a largo plazo, se trató de llevar a cabo de manera que dicho sistema de comunicación fuese fácilmente integrable en cualquier desarrollo IoT ya llevado a cabo hasta el momento en cualquiera de los proyectos activos. De ese modo, cuando estuviese listo el nuevo sistema, se evitaría el tener que abordar cambios complejos en despliegues IoT, que requiriesen rehacer partes sustanciales del *software*.

Por tanto, el desarrollo de IoTsafe, debía resultar en un método de comunicación lo más transparente posible a los elementos IoT ya en funcionamiento y, a su vez, debía lograr que el sistema que lo conformara favoreciese su despliegue y la conectividad para permitir una gestión ágil y segura de las infraestructuras IoT. Dado que las aplicaciones IoT utilizan internamente HTTP y otros protocolos sin seguridad para sus comunicaciones internas, se concluyó que sería preferible que la seguridad de las comunicaciones se desacoplara de las aplicaciones, para poder funcionar de forma transparente. Así, la arquitectura de IoTsafe que se presenta en este trabajo, se deriva de la utilizada inicialmente por los diseños llevados a cabo en elementos DS, que a su vez se inspiran en cierto modo en la filosofía de desacoplo MVC.

IoTsafe utiliza desarrollos de comunicación basados en proxificación, elaborados con el fin de proveer de conectividad y simplificar el despliegue, configuración y control de los dispositivos IoT. El esquema de comunicación por proxificación no contemplaba inicialmente el concepto de contexto de seguridad y ofrecía una protección limitada [109]. Fue en 2018, a partir de la

[73] Tutorial: Creación de una API web con ASP.NET Core, https://docs.microsoft.com/es-es/aspnet/core/tutorials/first-web-api

necesidad de escalabilidad e interconectividad, cuando el esquema se definió por completo, incluyendo nuevos conceptos (contextos de seguridad, *socket* protegidos, etc.), que permitieron elaborar un método y sistema de comunicación que implementase el desacoplo de seguridad y que a su vez fuese mucho más seguro y escalable. Esto dio lugar a la solicitud de su patente [110].

No obstante, el primer esquema de comunicación por proxificación [109] permitió estudiar la viabilidad de esta idea mediante pequeñas pruebas de concepto, así como la calidad de servicio ofrecida en las comunicaciones proxificadas y la disponibilidad de los servicios funcionando en entornos inalámbricos en movimiento (3.5G). Posteriormente, a partir de 2018 y debido a la necesidad de profundizar más en la seguridad de las comunicaciones, se desarrolló propiamente IoTsafe, incluyendo su concepto fundamental: el contexto de seguridad manejado desde el sistema operativo y transparente a la aplicación IoT. Para ilustrar el avance en el desarrollo de esta tecnología, se presentan varios precursores de IoTsafe, que fueron motivados por problemas reales, y se desarrollaron y llevaron a cabo junto con estudios experimentales que permitieron extraer conclusiones relativas a la factibilidad de la propuesta de comunicación finalmente presentada como IoTsafe.

3.1.1. **Sistema Digital Signage de exteriores**

El primer ejemplo de dispositivo DS desarrollado fue un tótem audiovisual de exterior. El diseño de la tecnología DS usada en el caso presentado siguió el patrón MVC. Este planteamiento se justifica por el hecho de que en casi todos los proyectos DS existe muy claramente una Vista (normalmente la pantalla), un Modelo (una base de datos con la información) y un Controlador (el programa desarrollado para la gestión y funcionamiento del sistema). La estrategia seguida consistió en lograr un desacoplo tecnológico entre estas tres partes para facilitar la reutilización del *software* desarrollado en diferentes proyectos. Para ello, en dicho proyecto se utilizó:

- **Controlador**: Un *software* consistente en un conjunto de utilidades y *script* Linux del sistema (*Bash*) y *software* en PHP capaz de comunicarse localmente vía HTTP con otros módulos *software,* y de realizar llamadas al sistema estándar, y comunicación con dispositivos periféricos (vía comandos para las interfaces *Recommended Standard* 232 o RS232[74], *High Definition Media Interface-Consumer Electronic Control* o HDMI-CEC[75], *General Purpose Input Output* o GPIO) y *Serial Peripheral Interface* o SPI).
- **Vista**: Explorador web *stand-alone* basado en Mozilla Firefox que reproduce páginas web con soporte de aceleración por *hardware*. Dicho explorador es independiente del sistema operativo Linux empleado y dispone de los controladores necesarios para

[74] Brian Peterson, "Serial Communications Jinni", *Sourceforge*, http://sjinn.sourceforge.net/example_send_hex.html
[75] Pulse-Eight, "USB CEC Adapter communication Library", *GitHub*, https://github.com/Pulse-Eight/libcec

interactuar con la pantalla táctil *multitouch*. El objeto de este *software* es manejar la carga de documentos web utilizando únicamente peticiones locales TELNET estándar, por medio del *plugin* MozRepl[76].

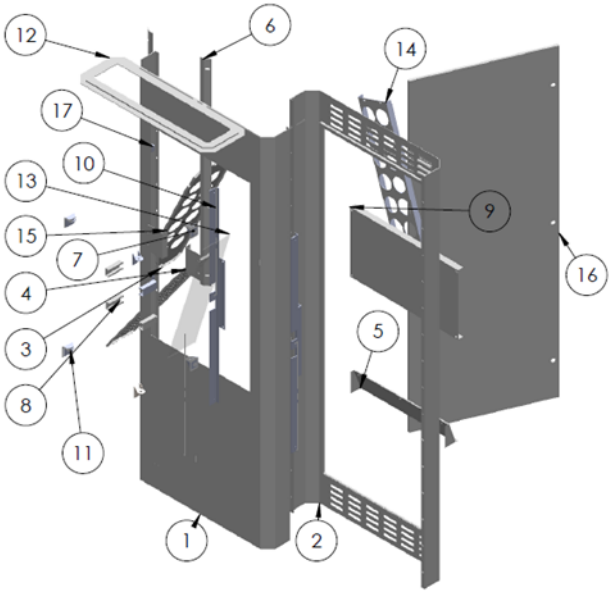
- **Modelo**: Una base de datos local en SQLite3 [77].

Los tres elementos *software* son independientes, y las comunicaciones entre ellos se llevan a cabo mediante llamadas al sistema o peticiones de red locales.

El dispositivo DS presentado en este ejemplo y conocido como "tótem de exterior" (Fig. 3.1-6) integraba varios elementos de interés: una membrana táctil capacitiva, un pantalla de 55" de alta luminosidad de 3.000 cd/m2 (diseñada específicamente para este caso), un sistema de ventilación especial por cortinas de aire para garantizar una refrigeración adecuada del sistema y pantalla, y un circuito de alimentación de 24 V. de alta potencia, controlado por relé. El diseño del producto se llevó a cabo en Servicios de TI de Durango como parte de un proyecto cofinanciado por el Consejo Nacional de Ciencia y Tecnología de México (Conacyt)[78].

En este proyecto se buscó además aprovechar la ubicación propia exterior del tótem para incorporar en él un dispositivo complementario que sirviera para monitorizar de forma cualitativa la calidad del aire. Para ello, se aprovechó el sistema de refrigeración por aire instalado en el tótem para incluir un módulo de medición de la calidad del aire exterior. Este dispositivo secundario consistió en una AirPi[79], un *shield* o una placa de expansión con múltiples sensores integrados para la plataforma RasperryPi. Su *software,* desarrollado en Python, se consiguió integrar con éxito inicialmente como parte de un dispositivo DS secundario que quedó conformado por la RaspberryPI y el AirPi [111]. Este dispositivo DS se considera secundario y posteriormente se integró con el tótem Luminia, el dispositivo DS primario.

[76] Massimiliano Mirra, "MozRepl plugin", *GitHub*, 2017, https://github.com/bard/mozrepl/wiki
[77] "SQLite Version 3 Overview" 2007, https://www.sqlite.org/version3.html
[78] Servicios de TI de Durango SA de CV, "Diseño y perfeccionamiento de una red Digital Signage para comunicación audiovisual en exteriores", *Programa de estímulos a la innovación*, 2014, https://www.conacyt.gob.mx/index.php/transparencia/transparencia-focalizada/fichas-publicas/2014-7/7297-213823-ficha-publica/file
[79] A. Dayan and T. Hartley, "AirPi", 2012, http://airpi.es/



ID	Descripción	Cantidad
1	Frontal del gabinete	1
2	Parte posterior del gabinete	1
3	Base de ventiladores	1
4	Soporte de ajuste invertido	1
5	Soporte angular a 70º	1
6	Soporte del vidrio	2
7	Soporte angular lateral	2
8	Perfil de ajuste horizontal	4
9	Tapa trasera	1
10	Soporte de tapa trasera	2
11	Soporte de ajuste	22
12	Tapa superior	1
13	Vidrio templado anti-reflejante	1
14	Soporte anterior de ventiladores techo	1
15	Soporte posterior de ventiladores techo	2
16	Puerta	1
17	Soporte de pantalla 1	2
18	Soporte de pantalla 2	2

Fig. 3.1-6: Despiece del Tótem Luminia.

Fig. 3.1-7: Interior y exterior del tótem Luminia en su versión táctil.

El patrón de arquitectura de este dispositivo secundario es similar al del tótem:

- **Controlador**: Consiste en el *software* desarrollado en Python para el control de la AirPi, junto con un sencillo *middleware* elaborado en PHP capaz de presentar la información de los sensores en formato web.
- **Vista**: Explorador web *stand-alone* elaborado en *webkit* QT5 (Fig. 3.1-8) capaz de reproducir páginas web y con soporte de aceleración por *hardware* para Raspberry[80,81]. Este *software* puede manejar la carga de documentos web utilizando únicamente peticiones HTTP estándar locales desde otro proceso diferente, comportándose como un explorador web controlable remotamente mediante comandos sencillos. Una parte de este software fue desarrollado en la tutela de un proyecto de estancia en la Universidad Tecnológica de Durango (UTD) a lo largo de 2014 [112].
- **Modelo**: El modelo en este caso puede considerarse el propio *hardware* AirPi. No existe una base de datos como tal, y los datos son obtenidos por el controlador en tiempo real.

El esquema de desarrollo seguido permitió integrar fácilmente este sistema secundario (Fig. 3.1-8) con el sistema DS primario (el tótem de exterior). Para ello, el controlador del tótem debía acceder al controlador del dispositivo secundario para solicitar la información de los sensores. Todo ello resultó factible utilizando únicamente comunicaciones simples basadas en HTTP en una red local aislada entre ambos dispositivos DS.

Fig. 3.1-8: Sistema secundario: Módulo AirPi V 1.0 funcionando conectado a una RaspberryPi.

[80] Luca Carlon, "Hardware Accelerated QtMultimedia Backend for Raspberry Pi and OpenGL Shaders on Video", *BlogSpot*, 2013,http://thebugfreeblog.blogspot.com/2013/04/hardware-accelerated-qtmultimedia.html
[81] Luca Carlon, "PiOmxTextures (POT) 5.7.0 with Qt 5.10.0 built for armv8 with GCC 6.3.0 on Raspbian Stretch", *BlogSpot*, 2018, http://thebugfreeblog.blogspot.com/2018/01/pot-570-with-qt-5100-built-for-armv8.html

El sistema de comunicación utilizado en estos dispositivos [109] se describe en la Fig. 3.1-9: se empleó SSH para proxificar las comunicaciones entre el dispositivo IoT y el servidor, lo que permite que la plataforma IoT, o incluso una aplicación de gestión externa, acceda a cada dispositivo fácilmente gracias a esas proxificaciones locales. De esta forma se evita tener que introducir configuraciones especiales en *firewall*, módems y *router* de las instalaciones en las que el dispositivo IoT fuese a operar. También habilitó la comunicación directa entre el dispositivo de control y el dispositivo IoT, o incluso entre dispositivos IoT (Fig. 3.1-10).

Sin embargo, pese a que todas estas prestaciones técnicas son factibles, este diseño no incluía propiamente un esquema de seguridad para el servidor o el propio *gateway*/DS *player* que permitiese garantizar que un dispositivo infectado no causase una reacción en cadena hacia otros dispositivos. Esto ocurría por ejemplo con algunos dispositivos *Zigbee* [113], en los que un dispositivo infectado ponía en riesgo toda la red. La estrategia que se siguió para mitigar este problema consistió en utilizar servidores IoT pequeños y dedicados, que gestionaran únicamente los dispositivos IoT que necesitaban manejar para ese proyecto en particular, y proteger el servidor con reglas firewall del exterior genéricas. Sin embargo, estas medidas sólo paliaban en parte el problema: si un dispositivo resultaba comprometido, podría conectarse hacia otros

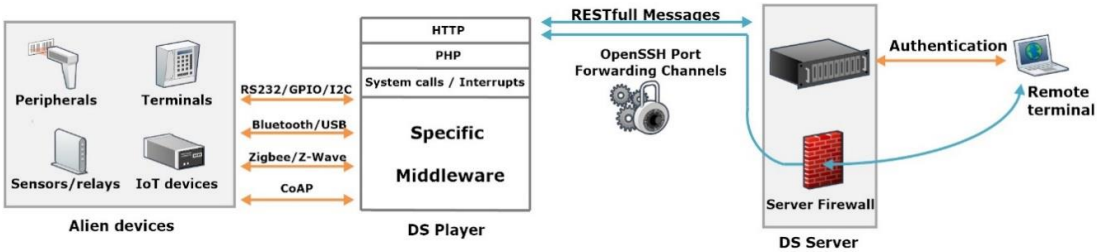


Fig. 3.1-9: Integración de dispositivos IoT utilizando un player DS como Gateway [136]**.**

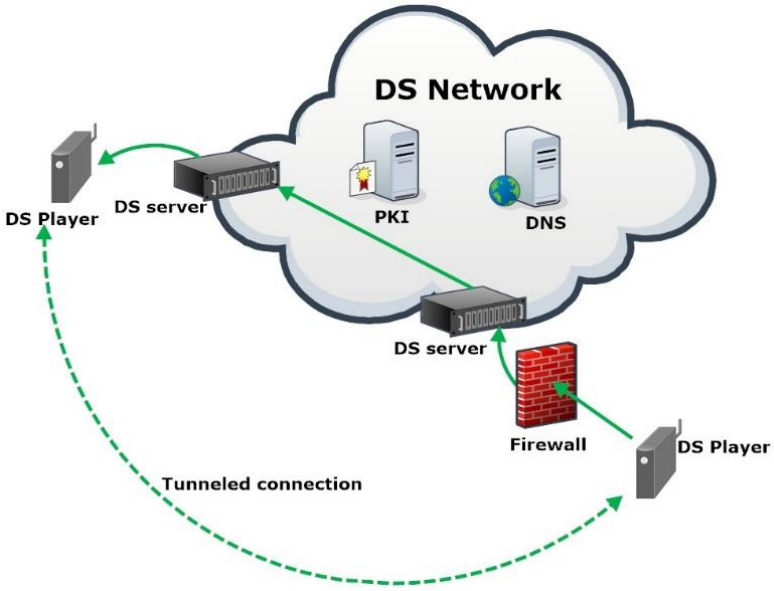


Fig. 3.1-10: Capacidad M2M de dispositivos DS utilizando comunicaciones proxificadas gateway [136]**.**

dispositivos a través de las proxificaciones locales existentes en el servidor, ya que éstas consistían en *socket* simples que ofrecen los servicios de los otros dispositivos sin ningún tipo de seguridad. Resultaba necesario diseñar un sistema de seguridad que, sin limitar la conectividad y flexibilidad ya conseguida, permitiese mejorar notablemente la seguridad para evitar este tipo de problemas.

3.1.2. **Sistema Digital Signage móvil para autobuses**

Los sistemas DS en autobuses permiten integrar el servicio de seguimiento del vehículo, las paradas, mensajes institucionales y publicidad en un mismo elemento. Este tipo de dispositivo debe emplear una electrónica suficientemente fiable para poder operar sin problemas en entornos hostiles (con vibración, movimiento, calor, alimentación eléctrica fluctuante, etc.).
Partes del diseño de este dispositivo se llevaron a cabo como una tutela de proyecto de estancia en la Universidad Tecnológica de Durango (UTD) a lo largo de 2016 [114], utilizando la plataforma RaspberryPi y el mismo patrón de arquitectura que el usado en el caso anterior:

- **Controlador**: incluye distintos *script* PHP y de shell que interactúan con el modelo y con el sistema GPS del dispositivo mediante simples llamadas al sistema. Este *software,* similar al de los tótems, guarda ciertas similitudes con un sencillo *Enterprise Resource Planning* (ERP) que gestiona en este caso los contenidos audiovisuales. Estos contenidos pueden ser imágenes, vídeos, pequeñas animaciones o sencillas aplicaciones web que proporcionen los servicios de información y geo-posicionamiento. El *software,* desarrollado en Python para su control junto con un sencillo *middleware* elaborado en PHP, es capaz de presentar la información de los sensores en formato *web*.
- **Vista**: Se reutiliza el explorador web *stand-alone* basado en QT5 que reproduce *web* sencillas, y que fue utilizado en el sistema secundario para la AirPi del proyecto anterior. Las páginas web soportadas pueden incluir distintos tipos de contenidos audiovisuales e información de interés relativa al trayecto del vehículo. El desarrollo de este tipo de contenido se considera muy sencillo gracias a la versatilidad de HTML5 y las tecnologías *software* asociadas (Framework Java*script*, CSS3, WebGL, WebCL, WebSockets, WebRTC, etc.).
- **Modelo**: El modelo consiste nuevamente en una base de datos SQLite3.

En este caso, el dispositivo DS se diseñó utilizando una RaspberryPi como base *hardware* (Fig. 3.1-11). Esto permitió su integración en los autobuses de forma más sencilla, al ocupar menos espacio y necesitar únicamente un zócalo DIN-1 (Fig. 3.1-12) equivalente al que utiliza un equipo estéreo estándar.

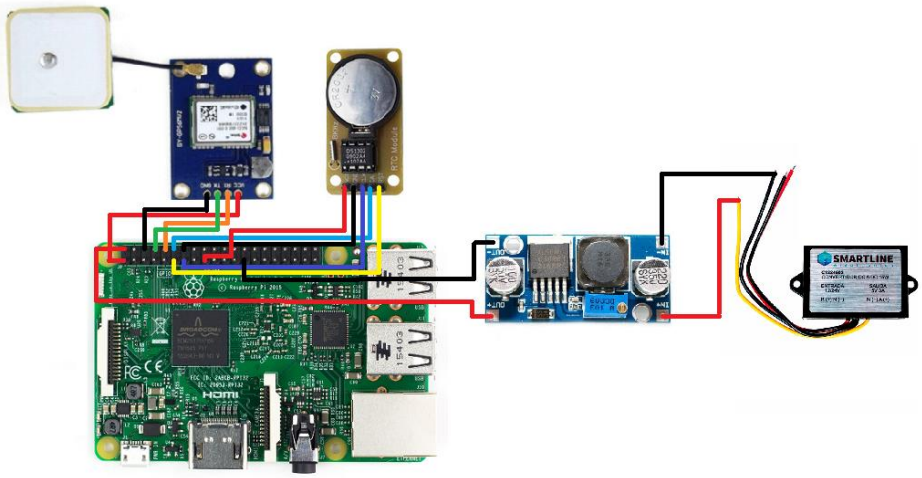


Fig. 3.1-11: ***Diagrama de conexiones de los dispositivos electrónicos del módulo Digital Signage móvil*** [114]***.***

Fig. 3.1-12: ***Instalación del player en un contenedor estándar DIN-1. Este dispositivo ya incluye la batería de ión de lítio de respaldo y el módem de banda ancha por Universal Serial Bus (USB).***

El sistema de alimentación, la carcasa y las salidas de audio y vídeo por interfaz HDMI y vídeo compuesto facilitaron la instalación en cualquier vehículo sin necesidad de conocimientos avanzados, siendo posible llevarla a cabo sin necesidad de un técnico (Fig. 3.1-13).

Este proyecto presentaba cambios sustanciales respecto proyecto del sistema DS para exteriores. La RaspberryPi, que está basada en tecnología ARM, tomó el rol del dispositivo DS primario que, en los tótems de exterior, era cubierto por *players* DS con de tecnología X86. Sin embargo, gracias al patrón de arquitectura MVC utilizado, gran parte de los desarrollos fueron reutilizables. Tras adaptar el sistema operativo Raspbian con únicamente los paquetes básicos necesarios, únicamente hubo que sustituir en la vista el explorador *custom* basado en Firefox,

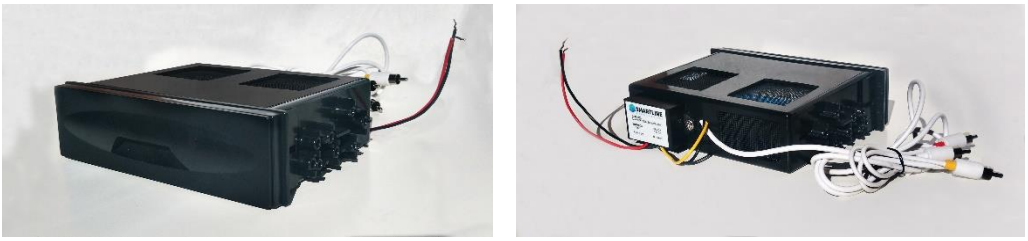


Fig. 3.1-13: Aspecto exterior del dispositivo Digital Signage móvil terminado.

por uno nuevo diseñado a medida de la Raspberry basado en QT5 que permitiese el control remoto del mismo. Este explorador es una evolución del utilizado inicialmente para la Airpi de los tótems de exterior, y permite un control remoto local (desde otro proceso de la misma Raspberry) de manera que la integración con el Controlador resulte similar a la del caso anterior, de cara a manejar contenidos audiovisuales, plantillas y aplicaciones HTML5/PHP. Los contenidos HTML5 dinámicos basados en aplicaciones web utilizados en el caso de la red DS de exterior son por tanto compatibles con estos nuevos dispositivos, consiguiendo así una reutilización de *software* efectiva.

Sin embargo, ambos casos (el que estamos presentando y el de sistema DS para exteriores) siguen siendo vulnerables a ataques desde un dispositivo infectado de la red DS hacia otros de la misma red, o desde potencial *malware* existente en el servidor. Estos riesgos se consiguieron minimizar definiendo un único servidor para tipo de dispositivo y proyecto, para así poder contener la expansión de potenciales vulnerabilidades entre clientes distintos.

3.1.3. **Sistema RFID de seguridad para control de personal**

Este proyecto se desarrolló como parte de la tutela de un proyecto de estancia en la Universidad Tecnológica de Durango (UTD) a lo largo de 2016 [115]. En dicho proyecto, se planteó el desarrollo de unos dispositivos RFID (Fig. 3.1-14) basados en el *hardware* D-Thik 501 V.3.2 (Fig. 3.1-15) para el control de acceso a la planta de ciclo combinado de Gas Natural Fenosa (ahora Naturgy) que dicha empresa posee en Durango, México.

Este tipo de plantas industriales deben realizar labores periódicas de mantenimiento, en las que se produce una revisión general de la maquinaria y equipamiento principal. Durante dichas paradas, la presencia de personal externo en planta hace que se incremente hasta en un factor de 10 el número total de trabajadores en las instalaciones. Por motivos de seguridad para la empresa, y por regulación laboral, resulta imprescindible controlar el acceso a cada una de las secciones de dichas instalaciones.

En este proyecto se desarrolló un dispositivo portátil capaz de leer las identificaciones RFID, para validarlas contra una base de datos de personal tanto interno como externo. Debía ser capaz de clasificar las distintas zonas de trabajo, y además poder funcionar en “modo emergencia” para poder comprobar, en caso de evacuación, quién pudiese quedar en la planta.

Fig. 3.1-14: Dispositivo RFID en funcionamiento.

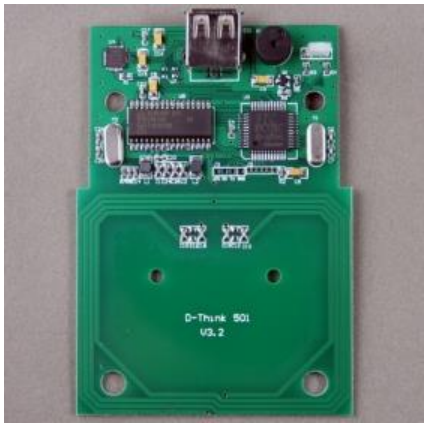


Fig. 3.1-15: Módulo RFID D-Thik 501 V.3.2.

El desarrollo se llevó a cabo con una plataforma Raspberry Pi 3 modelo B V1.2, una placa lectora RFID y un punto de acceso Pepwave Surf on-the-go[82]. La integración física (Fig. 3.1-16) se realizó en una carcasa (Fig. 3.1-17) diseñada en Solidworks y realizada mediante impresora 3D.

Los dispositivos pueden mostrar la información automáticamente en cualquier monitor por HDMI o inalámbricamente por medio de un dispositivo externo complementario (un teléfono o *tablet*) a través de una aplicación basada en PHP/HTML5 y SQLite. En este caso, el usuario se conectaba directamente al dispositivo utilizando una red WiFi de corto alcance y propia de cada uno de ellos que funciona como una *Wireless Personal Area Network* (WPAN). La aplicación de los dispositivos RFID portátiles se comunica con la interfaz *hardware* RFID a través de mensajes RS232 por medio de llamadas al sistema operativo.

[82] PEPWAVE Surf On-The-Go, https://download.peplink.com/resources/Pepwave_Surf_On-The-Go_Datasheet.pdf

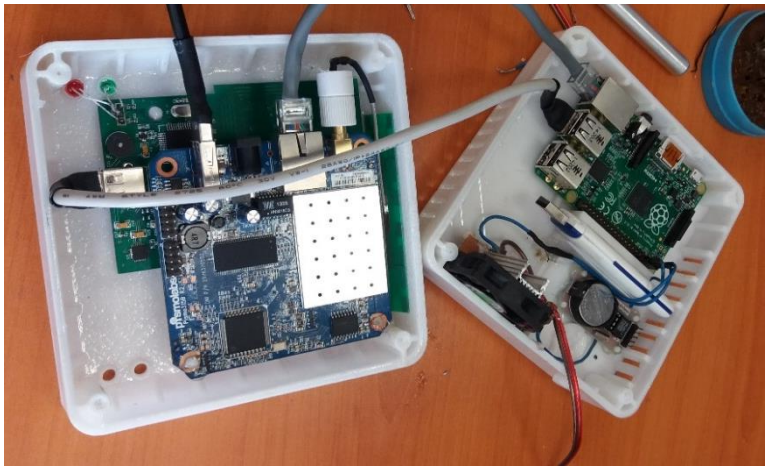


Fig. 3.1-16: Hardware interno del dispositivo RFID.

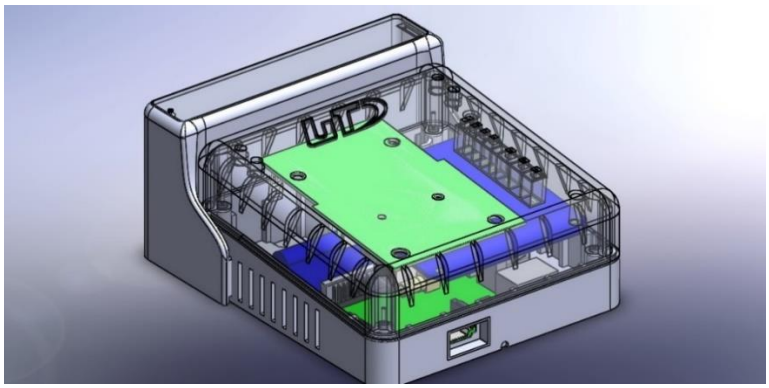


Fig. 3.1-17: Diseño solidworks del dispositivo lector RFID portátil con porta batería y salida HDMI.

El *software* utilizado en este caso siguió el mismo patrón MVC de los proyectos anteriores:

- **Controlador**: Consiste en distintos *scripts* de sistema (Bash) y PHP, que interactúan con el modelo y con el módulo RFID del dispositivo mediante simples mensajes RS232 sobre una interfaz serie virtual que opera a través de USB a nivel de sistema operativo. El controlador, además, gestiona la información referenciada por las tarjetas RFID a nivel local (datos de usuario, ubicación, permisos, etc), funcionando como un ERP ligero similar al que se utiliza en los tótems para manejar los contenidos audiovisuales.
- **Vista**: Se reutiliza el explorador web *stand-alone* basado en QT5, diseñado para Raspberries, explicado en los proyectos anteriores. La información que muestra la vista (bien sea local o remota) consiste en sencillas web en las que se identifica al trabajador propietario de la tarjeta, la hora de registro de entrada y la función de ingreso o salida efectuada junto con su fotografía. La vista puede utilizarse para administrar el controlador localmente con una tarjeta RFID especial o accediendo con el móvil o *tablet* a través de la WiFi local que cada dispositivo genera y emplear unas credenciales de administración.
- **Modelo**: El modelo consiste nuevamente en una base de datos SQLite3.

El modo de funcionamiento del sistema incluía las funciones necesarias para operar como control de acceso en condiciones normales y de emergencia:

- Todos los dispositivos portátiles operan siempre a nivel local con autonomía por batería externa, y también de forma remota si hay conectividad 3G o de red.
- Existe un servidor local al que se puede acceder vía Internet o vía red local, al que los dispositivos portátiles se conectan para mantener la información actualizada de ubicación de las tarjetas RFID registradas.
- En caso de emergencia, cualquier dispositivo puede convertirse en un validador de personal evacuado utilizando cualquier tarjeta RFID de personal autorizado. En ese caso, el dispositivo verifica si existe alguna diferencia entre la base de datos local y la remota, y procede a operar en modo emergencia: Toda tarjeta RFID de personal detectada a partir de entonces sirve para determinar que dicha persona ha salido de la planta y se encuentra en un punto exterior seguro de reunión. La información de dicho proceso de evacuación se podía monitorizar con una tablet o móvil, conectándose directamente al dispositivo a través de la WiFi generada por el propio dispositivo.
- Los dispositivos, a su vez, tratan de volcar siempre que pueden cualquier información nueva del servidor local en su base de datos, y de actualizar la información de dicho servidor local con los datos nuevos que el propio dispositivo haya podido generar. Para ello:
    - Existe una adecuada sincronización de tiempos entre dispositivos.
    - La red local WiFi de la empresa que se utiliza para ofrecer conectividad a los dispositivos permite una conexión al servidor local.
    - En caso de fallar la red WiFi local, los dispositivos pueden conectarse entre ellos usando sus propias redes locales WiFi.
- En caso de no poder conectarse al servidor vía red local, con ayuda de infraestructura WiFi, los dispositivos pueden conectarse a un servidor remoto vía Internet, que se mantiene sincronizado con el servidor local haciendo uso del módem 3G que poseen.
- En caso de interrupciones en las comunicaciones, las discrepancias existentes entre las bases de datos de dispositivos, servidor local y servidor remoto se resuelven fácilmente gracias a la sincronización en tiempo de los dispositivos.

De nuevo, este proyecto como todos los anteriores puede considerarse como una prueba de concepto funcional. Sin embargo, si se pretende una solución escalable a otras plantas, así como aumentar el número de terminales lectoras, es necesario reforzar la seguridad en los dispositivos RFID y en los servidores para evitar problemas como los descritos anteriormente.

3.1.4. Necesidad de mejora en la seguridad de las comunicaciones

Los proyectos analizados en las secciones anteriores presentan una metodología de diseño y desarrollo de dispositivos y servicios bastante ágil, capaz de dar respuesta a problemáticas diversas con esquemas de *software* similares que permiten su reutilización. Sin embargo, cuando dichos sistemas requieren comunicaciones entre sí, con elementos externos, o se requiere escalar la solución, se aprecian problemas importantes en el manejo de la seguridad, y en la viabilidad de una gestión igualmente ágil de estos tipos de sistemas.

Los principales problemas de seguridad se centran en el servidor (y en el *gateway* si está presente). La estrategia de proxificación de comunicaciones resulta muy interesante y ofrece múltiples ventajas, pero requiere de un tratamiento complementario de la seguridad en el servidor que, de no producirse, pone en peligro toda la red. Por ello, surge la necesidad de buscar un esquema de comunicación que aporte la seguridad necesaria a los sistemas, sin añadir complejidad a los diseños ya implementados y que, a su vez, facilite la escalabilidad sin necesidad de incrementar la complejidad de su gestión.

En la Fig. 3.1-18 se muestran algunos de los ataques más importantes que pueden ocurrir en el servidor IoT que gestione comunicaciones proxificadas, si un dispositivo IoT de los que gestiona se encuentra comprometido. En esta figura se muestran cuatro dispositivos IoT que se comunican con el servidor IoT a través de proxificaciones conducidas por sesiones SSH. En este ejemplo, cada dispositivo no comprometido utiliza dos proxificaciones:

- **Proxificación directa**: Permite a la aplicación del dispositivo IoT acceder localmente desde el dispositivo al recurso ofrecido por la plataforma IoT. Este recurso se ofrece en el dispositivo IoT como *socket-proxy* directo.
- **Proxificación reversa**: Permite al dispositivo IoT poner a disposición de la plataforma IoT un recurso de la aplicación IoT del dispositivo. Este recurso se ofrece localmente en el servidor IoT como *socket-proxy* reverso.

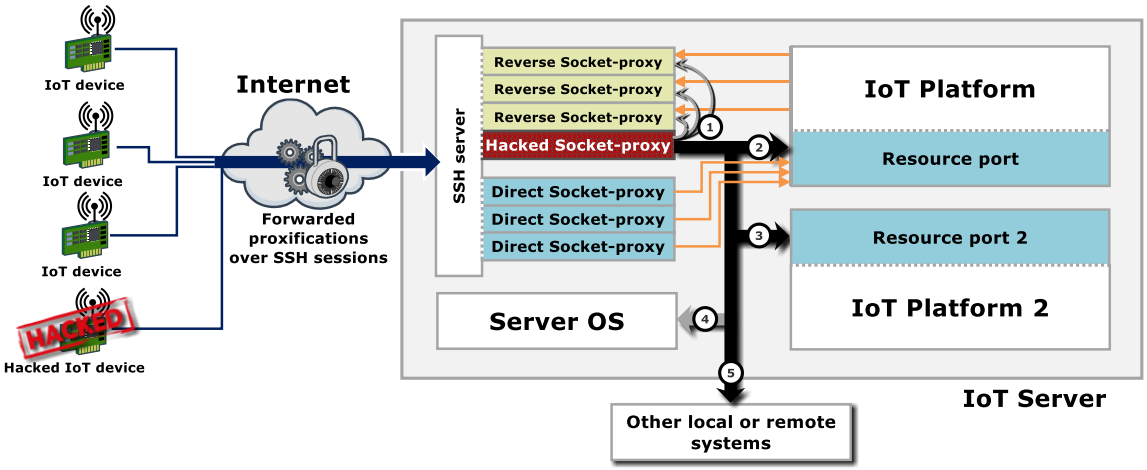


Fig. 3.1-18: Posibles ataques derivados de un dispositivo hackeado en una red IoT que utiliza proxificaciones en el servidor IoT basadas en SSH.

El dispositivo comprometido puede aprovechar las funcionalidades de SSH y sus credenciales de acceso al servidor IoT para desplegar en él un número indeterminado de *socket-proxies* reversos y directos, a través de los que podría realizar los siguientes ataques:

1) **Ataques hacia otros dispositivos IoT**. Aprovechando la existencia de otros *socket-proxy* reversos de otros dispositivos IoT, el dispositivo IoT comprometido podría tratar de acceder a esos dispositivos, con el objetivo de causar un funcionamiento erróneo en los aparatos, sustraer información de ellos o aprovechar vulnerabilidades. Es importante mencionar que estos *socket* se utilizan por protocolos de aplicación sin seguridad (delegada en SSH) por lo que resulta más sencillo realizar este tipo de ataques.
2) **Ataques hacia la plataforma IoT**: El dispositivo comprometido puede tratar de suplantar la identidad de otros dispositivos IoT y suministrar a la plataforma información errónea. También puede lanzar ataques de denegación de servicio a la plataforma, solicitando numerosas peticiones en paralelo. Si el servidor SSH no se encuentra configurado adecuadamente, el dispositivo comprometido podría lanzar un ataque hacia el servidor tratando de generar una gran cantidad de *socket-proxies* hasta agotar los recursos del servidor.
3) **Ataques hacia otras plataformas IoT**. Si el servidor IoT se encuentra compartido con otras plataformas IoT, estas podrían sufrir un ataque cruzado de un dispositivo comprometido de otra red IoT distinta. Sus dispositivos IoT, a su vez, también podrían ser vulnerables y sus *socket-proxies* ser atacados de igual manera que en el ataque 1. Si la plataforma IoT segunda es finalmente comprometida remotamente por un dispositivo IoT, esta otra plataforma podría tratar de comprometer más rápidamente sus propios dispositivos IoT mediante actualizaciones de seguridad con *malware*.
4) **Ataques hacia el sistema operativo**: Si el sistema operativo del servidor tiene algún tipo de servicio operando con *socket* de red, éstos son susceptibles también de sufrir ataques de dispositivos IoT comprometidos, a través de los *socket-proxy* reversos del dispositivo IoT comprometido. Esto puede degenerar en un escalado de privilegios aprovechando alguna vulnerabilidad conocida del servicio atacado, y llegar a conseguir acceso en claro a todo el tráfico local en el sistema. Esto sería posible gracias a que las comunicaciones entre los *socket-proxies* y las plataformas se producen utilizando protocolos inseguros. Aunque el servidor SSH esté configurado inicialmente para evitar sesiones de *shell*, es posible que al comprometer algún servicio del sistema operativo del servidor se pueda tener acceso al sistema, y de ahí a los servicios básicos que el sistema operativo ofrece y, si consigue escalar privilegios, podrían usarse con rango de administrador.

5) **Ataques a otros dispositivos locales**: El *socket-proxy* originado por el dispositivo IoT comprometido también podría lanzar ataques hacia otros dispositivos de la red local del servidor IoT o incluso remotos desde el propio servidor. Estos ataques podrían ser de múltiples tipos: *Mail spam*, DoS, *brute-force*, etc.

Estos riesgos se pueden mitigar notablemente con medidas sencillas habituales en la administración de infraestructuras de red:

- Limitar el acceso del tráfico procedente de direcciones IP locales del servidor. Estas direcciones IP se utilizan en los *socket-proxy* para comunicaciones estrictamente locales. Por ese motivo, resulta razonable establecer reglas de firewall estáticas que impidan todo tráfico desde/hacia direcciones IP locales con IP no locales. Esta medida solucionaría el ataque 5.
- Limitar el rango de direcciones aplicables a los *socket* proxificados y dedicarlas a ese fin de forma exclusiva. Esta medida, fácilmente implementable con una regla firewall estática, evitaría los ataques al sistema operativo desde *socket-proxies* de dispositivos comprometidos (ataque 3) y dificultaría un posible escalado de privilegios (ataque 4).
- Evitar compartir un mismo servidor con múltiples plataformas IoT. En ausencia de un mejor sistema de seguridad, lo más razonable es utilizar servidores pequeños dedicados para cada proyecto IoT en exclusiva, impidiendo así cualquier tipo de propagación de una vulneración de seguridad entre plataformas. Para evitar completamente el tipo de ataque 3, también se desaconsejaría la interconexión entre servidores usando proxificaciones. Siguiendo esta recomendación, no resultaría factible la horizontalización de las comunicaciones sin la adopción de medidas adicionales de seguridad.
- Los ataques número 1 y 2 se podrían mitigar en parte con reglas estándar de firewall que identifiquen ataques o cambios sustanciales en el comportamiento de las comunicaciones, apoyándose en técnicas de *Deep Inspection* (DI) basadas en Inteligencia Artifical (IA). Sin embargo, estas medidas se considerarían reactivas y necesitarían de ataques exitosos para empezar a operar.

El análisis de las vulnerabilidades existentes derivadas de una comunicación basada en proxificaciones sugiere que, aunque se pueden mitigar en parte con configuraciones de seguridad complementarias, el resultado de las mismas acaba siendo insuficiente. Se requiere un desarrollo específico en el servidor que no sea únicamente reactivo (mitigación del ataque) para los casos 1 y 2, aplicando en cambio un diseño seguro (*secure-by-design*) que trate de alcanzar mayor infalibilidad para ser más escalable y confiable. Además, la horizontalización de las comunicaciones entre plataformas IoT no parece factible únicamente con el diseño actual, por el crecimiento de la inseguridad en línea con el número de dispositivos conectados.

Por todo ello, es necesario elaborar un sistema de gestión de las comunicaciones proxificadas en el servidor, que permita un manejo adecuado de la seguridad, permisos de

conexión y acceso a los recursos ofrecidos. Todo ello debe ofrecerse de la forma más transparente posible, para evitar perder la flexibilidad inherente a las comunicaciones proxificadas, así como las ventajas del desacoplo entre las aplicaciones IoT y las funcionalidades de seguridad requeridas.

3.2. IOTSAFE: MÉTODO Y SISTEMA DE SEGURIDAD PARA ECOSISTEMAS IOT

Una vez analizados los sistemas precursores, en esta subsección presentamos la solución IoTsafe. Repasaremos para ello los objetivos de la investigación planteados en la sección 1.3. Como veremos, el uso de un sistema operativo como Linux, ya presente en múltiples soluciones de IoT, puede dar respuesta a algunos de los retos expuestos:

C1: ¿Se pueden abordar los problemas de seguridad de forma general, pero sin requerir por parte de los desarrolladores el uso de tecnologías de seguridad/framework específicos de forma obligatoria?

La manera en la que se plantea la programación MVC en la sección 3.1, en la que se confirma la utilidad de la independencia entre el Modelo, Vista y Controlador, requiere que el entorno de ejecución de las aplicaciones permita soluciones *stand-alone* con poca o nula interdependencia con otros módulos *software*. En la sección 2.1 se explica cómo las distribuciones Linux utilizan colecciones de paquetes y librerías que conforman el sistema operativo y permiten ahorrar espacio entre las distintas aplicaciones que comparten funcionalidades concretas, reusando código compilado común de forma recurrente. Sin embargo, Linux también permite el desarrollo de programas *stand-alone* sin dependencias de librerías (aunque ocupan más espacio). Tal y como se explica en esa misma sección (2.1), la compatibilidad a nivel binario de esta solución estaría garantizada en un amplio número de casos en los que dichas aplicaciones no requieran acceso directo a funcionalidades muy específicas de versiones concretas del *kernel*.

De ese modo, la seguridad de las aplicaciones para dispositivos IoT podría quedar delegada en un módulo de seguridad *stand-alone* (con mínimas dependencias del *kernel*), con el que las aplicaciones del dispositivo se comunicarían mediante tráfico de red local. Estas aplicaciones utilizan la API de sockets de red como interfaz de conexión con el módulo stand alone. Dado que la interfaz de sockets está universalmente extendida y resulta muy invariable, se garantizaría así una independencia entre las aplicaciones IoT y el módulo stand-alone de seguridad. En estas circunstancias, dicho módulo de seguridad podría diseñarse para operar de manera transparente de forma que cualquier actualización de una de las dos partes (aplicación y módulo de seguridad) no afectaría a la otra en ningún caso (Fig. 3.2-1).

Fig. 3.2-1: Actualización de seguridad de un dispositivo IoTsafe: El módulo de seguridad stand-alone en el dispositivo se comunica con la aplicación principal IoT mediante sockets de red únicamente, gracias a su diseño que permite la independencia de software [108]**.**

C2: ¿Cuál sería el esquema de seguridad técnico propuesto?

La solución técnica presentada como IoTsafe, consiste en un método y sistema (actualmente en proceso de patente [110]) que persigue aportar soluciones para los problemas del estado del arte abordados anteriormente. En esta subsección se explicarán los conceptos fundamentales de esa patente, por lo que se apoya de textos íntegros y figuras de la misma.

IoTsafe es un sistema de comunicación segura basado en proxificación de *socket* que permite a las aplicaciones de los dispositivos delegar las funcionalidades de seguridad en módulos de *software* independientes y *stand-alone,* que se puedan actualizar sin afectar al resto del *software* de los dispositivos. Este objetivo se consigue combinando una comunicación segura basada en proxificaciones entre dispositivos, con un control de las comunicaciones locales empleando *contextos de seguridad,* que permiten restringir el comportamiento y el acceso a dichas proxificaciones en función del origen de la comunicación. Esta estrategia permite simplificar enormemente el desarrollo de las aplicaciones y el mantenimiento de la seguridad a medio y largo plazo, aun cuando el soporte técnico de dichas aplicaciones deje de realizarse. Esto es posible gracias a que el sistema descrito por IoTsafe no forma parte de las aplicaciones y puede actualizarse sin afectarlas.

C3: ¿Se puede diseñar el sistema de seguridad de manera que dispositivos IoT arbitrarios puedan incorporarse al mismo? ¿Qué requisitos mínimos serían necesarios?

La posibilidad de que dispositivos arbitrarios puedan incorporar IoTsafe a su arquitectura de forma sencilla resulta factible si se cumplen algunos requisitos:

- Que el dispositivo IoT opere bajo Linux o un entorno POSIX que pueda ejecutar aplicaciones binarias *stand-alone*.
- Que el dispositivo IoT disponga en su *kernel* de algún módulo como Netfilter, que opere como *framework* de red y permita proteger el tráfico local entre aplicaciones.
- Que el dispositivo IoT incorpore algún tipo de protocolo de comunicaciones que permita realizar con garantías proxificaciones seguras entre máquinas.

De este modo, cualquier dispositivo que incorpore una versión de Linux con Netfilter y SSH cumpliría con los requisitos mínimos.

3.2.1. **IoTsafe en dispositivos con FreeRTOS**

Los requisitos mínimos para implementar IoTsafe, mencionados en la sección anterior, están orientativos para garantizar una implementación sencilla. En realidad, para una implementación generalista, tan sólo hace falta que exista cierta independencia entre la aplicación, el sistema operativo y los módulos de seguridad de IoTsafe, basados preferentemente en SSH.

Existen en el mercado dispositivos desarrollados a partir de microcontroladores con recursos de RAM y ROM del orden de las decenas y centenas de Kbytes. Son aparatos muy limitados, que no pueden implementar aplicaciones de la misma manera que en otro tipo de sistemas menos restringidos. Tal y como se describió en la sección 2.1, la ROM de estos aparatos suele contener únicamente un único binario de formato ELF, con el RTOS y la aplicación. Ambos elementos forman un bloque monolítico binario normalmente indivisible (BIN) que, para actualizarse, debe reemplazarse por completo, y normalmente de forma manual in-situ.

Existen, no obstante, algunas iniciativas que permiten que partes o la totalidad de este BIN puedan actualizarse, entre las que se encuentra el uso de un "*bootloader*". El *bootloader* es un *software* específico de cada arquitectura y de cada dispositivo en particular, y suele encontrarse en una región *de solo lectura* de la memoria ROM que no es actualizable. En este caso, el *bootloader* tan sólo inicializa el *hardware* del microcontrolador y lo prepara para poder ejecutar el binario que reside en la ROM del dispositivo.

En caso de que en la ROM cargada existia un segundo binario que comprendiese un RTOS con otra aplicación (Fig. 3.2-2), el *bootloader*, podría decidir durante el arranque cargar esta segunda aplicación en vez de la primera, simplemente apuntando a una dirección diferente de memoria en la que continuaría el proceso de arranque. Sin embargo, el *bootloader* que por defecto tiene el dispositivo no suele soportar este tipo de lógicas, por lo que se recurre a un *second stage boot loader* (*SSBL*) que permite esta funcionalidad. Este SSBL puede considerarse una aplicación muy sencilla, que realiza algún tipo de comprobación y decide qué aplicación[83] cargar. En la Fig. 3.2-2 se muestra un ejemplo de esta situación, en la que existen dos aplicaciones A y B. Para que esta estrategia funcione, este SSBL se carga antes que la parte del BIN que contiene el RTOS y la aplicación correspondiente. De ese modo, es posible realizar la lógica básica requerida de forma transparente, en la que se determina la aplicación que se debe cargar.

[83] En este caso se entiende la aplicación A o B como una parte del BIN cargado en el microcontrolador que incluye cada una su propio RTOS y su aplicación IoT. Esto quiere decir que la aplicación A incluye su propio RTOS y la aplicación B también.

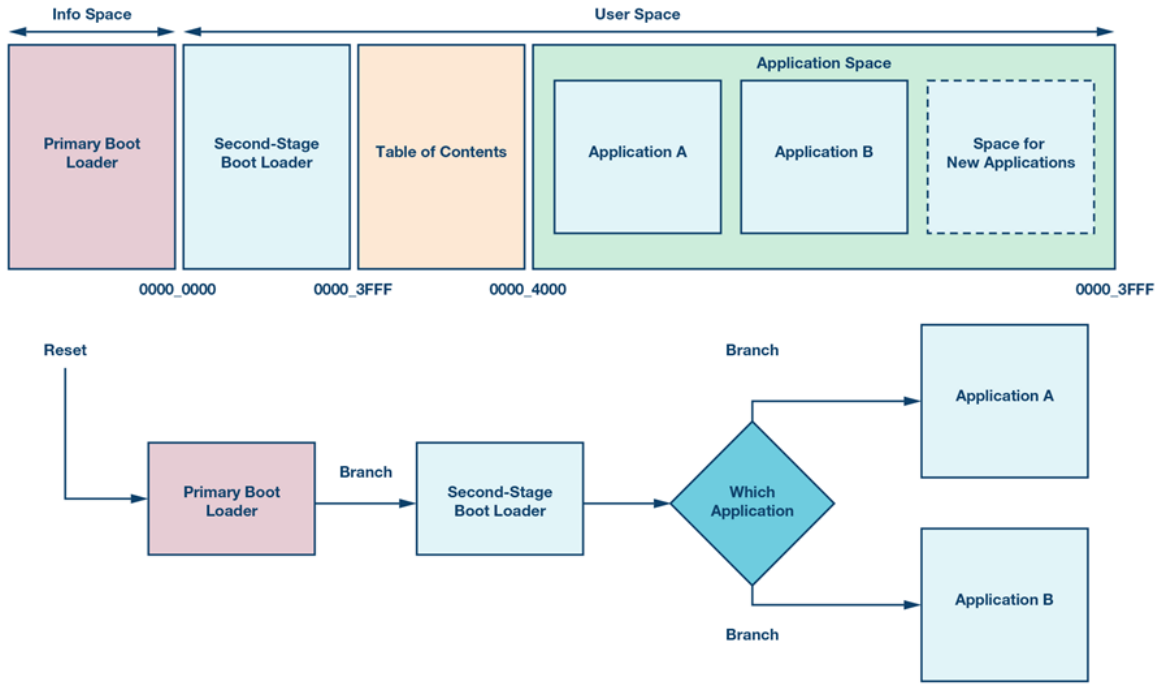


Fig. 3.2-2: Proceso de arranque de un microcontrolador con bootloader y SSBL [116]**.**

Teniendo este esquema presente, es posible diseñar un sencillo procedimiento que permita una actualización *Over The Air* (OTA) o actualización remota. Para poder realizar la actualización, se pueden emplear dos aplicaciones en el dispositivo:

- La aplicación A, que opera el dispositivo normalmente para cumplir con el objeto para el que fue diseñado.
- La aplicación B, utilizada para el proceso de actualización y que configuraría el SSBL para iniciar dicho proceso.

De este modo, el microcontrolador carga normalmente la aplicación A, y cuando se requiere realizar una actualización, esta aplicación altera el SSBL y reinicia el dispositivo para que cargue la aplicación B diseñada para realizar la OTA. Esta aplicación contendría la programación necesaria para iniciar un proceso de actualización de la sección de la ROM correspondiente a la aplicación A (Fig. 3.2-3). Una vez terminado el proceso, el dispositivo configuraría el SSBL para que cargue la nueva versión de la aplicación A y se reiniciaría para arrancar normalmente y funcionar con la aplicación actualizada.

Las aplicaciones (RTOS y aplicación IoT) deben compilarse de forma especial, para que cuando se genere el ELF/BIN final, las direcciones de memoria de las llamadas a funciones que comprendan, estén adecuadamente redireccionadas con su *offset* correspondiente. Esto lo debe hacer el *software* generador del ELF o BIN o el propio microcontrolador si soporta "*remapping the interrupt vector table*" automáticamente[84].

[84] NXP Semiconductors "LPC178x/7x User manual: 2.4 Memory re-mapping", 2016, https://www.nxp.com/docs/en/data-sheet/LPC178X_7X.pdf

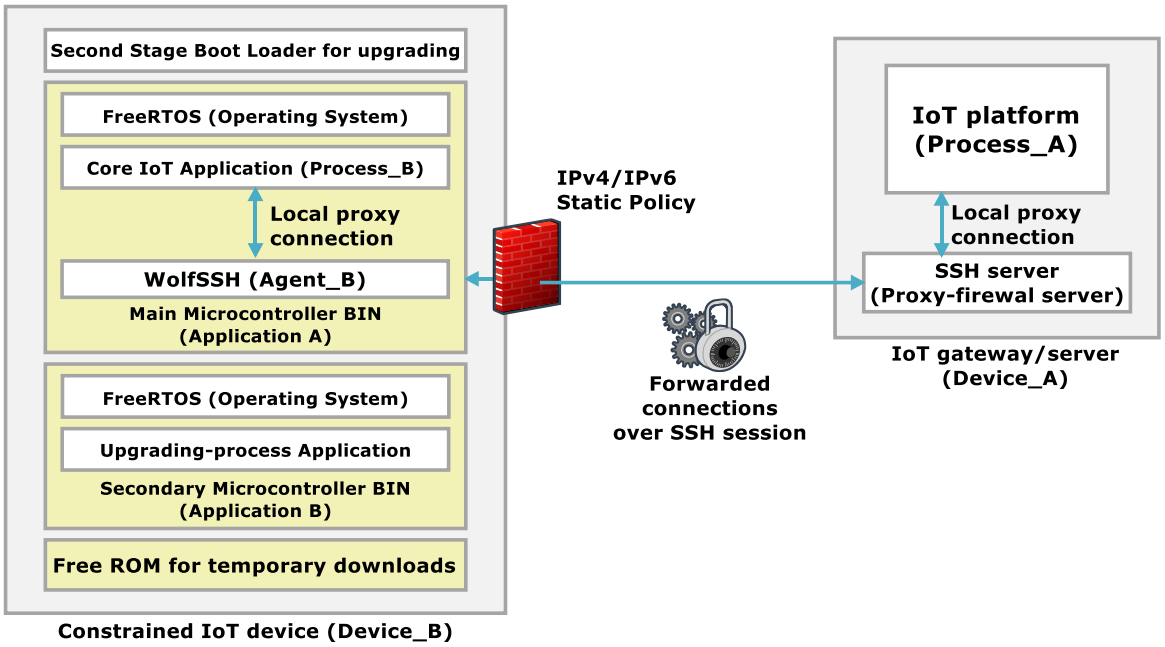


Fig. 3.2-3: ***Posible estructura de la ROM de un microcontrolador utilizando FreeRTOS y WolfSSH para implementar las comunicaciones proxificadas de IoTsafe.***

IoTsafe puede cumplir en este caso dos objetivos diferentes: ofrecer la funcionalidad de actualización para la aplicación IoT, y proporcionar una forma segura de comunicación en el proceso de actualización remota. Para ello, en caso de usar FreeRTOS, sería necesario que se utilizase incluyendo las funcionalidades de compatibilidad POSIX. De ese modo, si el microcontrolador cuenta con suficientes recursos, podría incorporar las funcionalidades SSH requeridas gracias a una versión del protocolo optimizada para entornos altamente restringidos, como la que ofrece WolfSSH (servidor/cliente)[85]. La parte de código correspondiente a WolfSSH no se puede actualizar directamente, al encontrarse ligada estáticamente en el BIN. Se debe de compilar el nuevo ELF/BIN con la nueva versión de WolfSSH y este BIN cargarlo de nuevo en el microcontrolador mediante actualización OTA. Sin embargo, aunque este *software* se integra en el mismo BIN que el RTOS y la aplicación, se puede considerar "*portable*" hasta cierto punto, ya que las actualizaciones de este software no van a requerir nunca de cambios en el código fuente del resto del *software* que conforma el BIN ya que la API que emplean para comunicarse no cambia en el tiempo: la API de sockets (*local proxy connecton*, Fig. 3.2-3).

Para poder realizar la OTA usando IoTsafe, se tendría que descargar una versión completa de la nueva imagen BIN de la aplicación A (Fig. 3.2-4) y este proceso lo manejaría el Agente_B. Esta nueva versión de la aplicación A se quedaría de forma temporal en el espacio libre de la memoria ROM. Posteriormente, se configuraría el SSBL para que cargase la aplicación B encargada de la actualización (no haría falta que se conectase a Internet, ya que tiene la nueva imagen descargada) y simplemente sobrescribe la aplicación A con su versión actualizada

[85] WolfSSH, "wolfSSH Lightweight SSH Library", https://www.wolfssl.com/products/wolfssh/

previamente descargada. Esta estrategia permite que la SSBL sea mucho más sencilla (no requiere tener el WolfSSH) y ocupar menos espacio en la ROM de forma permanente, aunque para la actualización es necesario reservar el espacio suficiente para la descarga.

Por tanto, aunque en los RTOS se puede lograr un desacoplo de seguridad a nivel conceptual con IoTsafe, sí es necesario actualizar el ELF/BIN principal del microcontrolador de forma completa en caso de actualización de seguridad, debido a la limitación de FreeRTOS y su *static-linking*. Sin embargo, estas actualizaciones son muy sencillas de hacer, no requieren conocimiento alguno de la programación de la aplicación principal de la aplicación IoT, ni se requieren modificaciones a la misma en ningún momento, pudiéndose realizar de forma automática. Este grado de independencia a nivel de código fuente es posible porque la "*Core IoT Application*" y WolfSSH se comunican localmente por mensajes de red y son totalmente independientes.

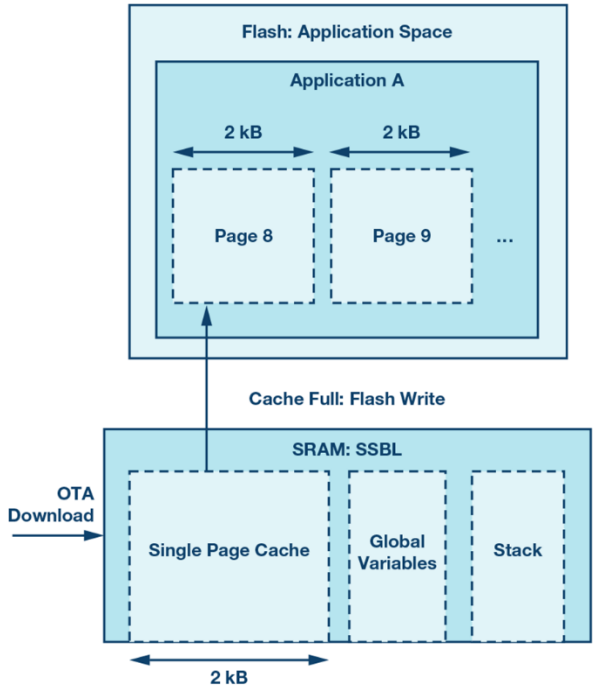


Fig. 3.2-4: ***Proceso de actualización OTA de un microcontrolador con bootloader usando caché parcial en RAM*** [116]**.**

3.2.2. **Contextos de seguridad**

El sistema IoTsafe utiliza proxificaciones para "acercar" *sockets* del dispositivo IoT al servidor o viceversa, y que de ese modo las aplicaciones IoT puedan conectarse a ellos localmente sin tener que preocuparse de la seguridad de los protocolos de comunicación involucrados. Este esquema es aprovechado a su vez por la plataforma IoT para manejar de forma ágil y segura las comunicaciones con sus dispositivos IoT (Fig. 3.2-5).

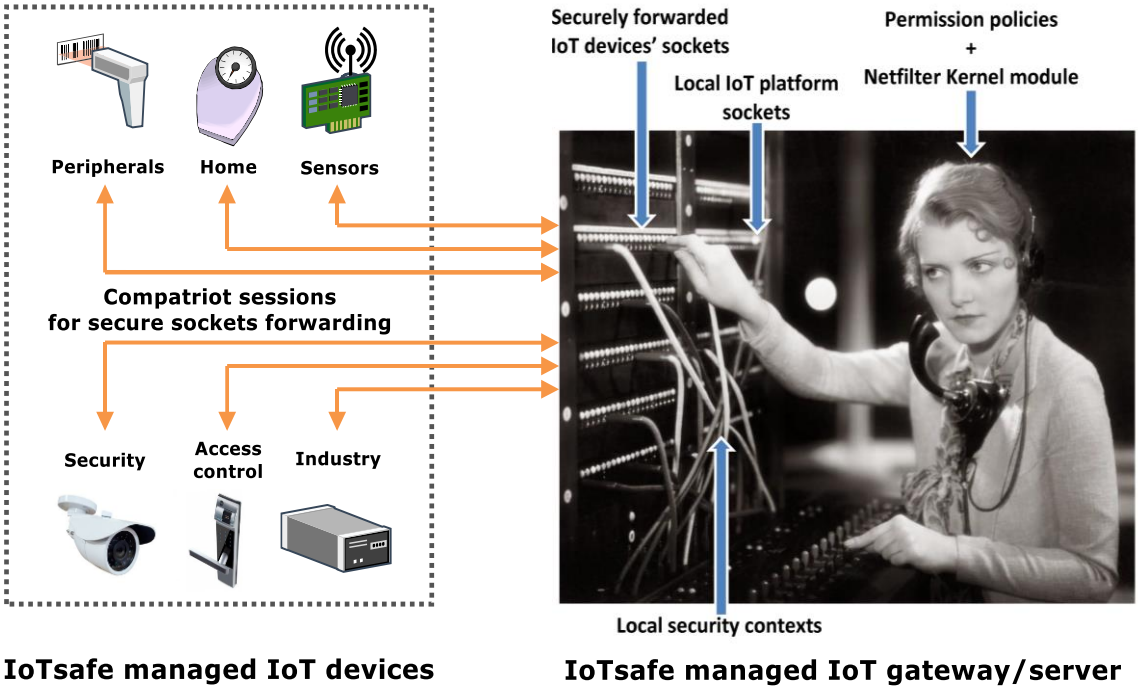


Fig. 3.2-5: Esquema simplificado de un servidor/gateway de IoTsafe en el que las comunicaciones se representan como enlaces característicos de un sistema de conmutación de circuitos telefónicos.

Esta estrategia presenta ciertas similitudes con las redes MPLS para redireccionar tráfico IP de un extremo a otro extremo de la red. En el caso de MPLS, cada paquete IP recibe una etiqueta que se le añade al ingresar en la red. Este paquete circula por la red en base a la etiqueta recibida y al dejar la red MPLS pierde la etiqueta (Fig. 3.2-6). El uso de la etiqueta en vez de la dirección IP agiliza el envío de paquetes y facilita el redireccionamiento de tráfico en tiempo real dentro de la red.

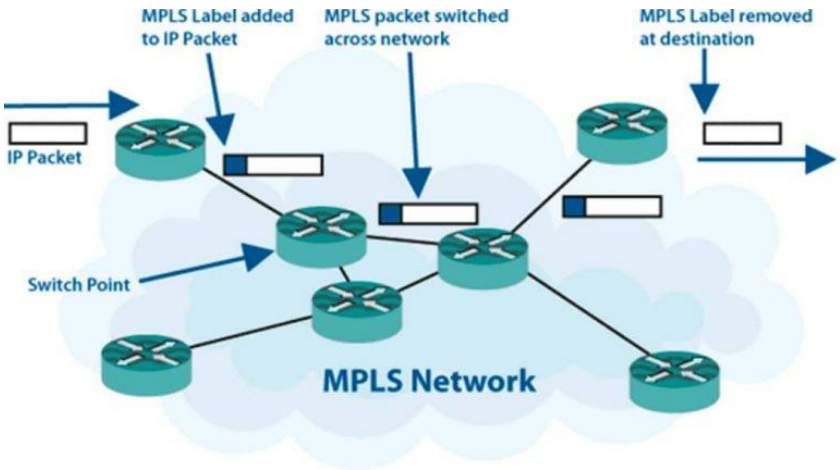


Fig. 3.2-6: Esquema de funcionamiento de una red MPLS para redireccionar el tráfico con etiquetas[86].

[86] Mushroom networks, "https://www.mushroomnetworks.com/what-is-mpls/"

Como se verá posteriormente, IoTsafe guarda cierto paralelismo con las redes MPLS: utiliza tráfico IP, emplea etiquetas (campo TOS) y en sus objetivos se incluye el control del flujo de tráfico (a través de reglas de marcado y filtrado del *framework* de red). Sin embargo, mientras que MPLS está pensado para un ámbito de red, el marcado de IoTsafe sólo tiene significado local dentro de la propia máquina. Además, la motivación detrás de este marcado es el de gestionar los permisos de accesos a recursos protegidos (*sockets* locales), mientras que en MPLS normalmente se utiliza para equilibrar la carga de la red y realizar ingeniería de tráfico.

El concepto de "contexto de seguridad" presentado por IoTsafe y que da sentido al esquema de marcado y filtrado de tráfico interno, surge a partir de la necesidad de dar respuesta a la cuestión C4 planteada en la sección 1.3:

C4: ¿Es factible integrar el esquema de seguridad en una infraestructura de comunicación horizontal que permita una comunicación de forma sencilla entre elementos de diferentes verticals?

Antes de que se plantease en IoTsafe el uso de los contextos de seguridad, las comunicaciones proxificadas de los ejemplos precursores de la sección 3.1 resultaban altamente inseguras, particularmente del lado del servidor. Por ello, se optaba por dedicar únicamente un servidor de dispositivos IoT por proyecto, juntamente con reglas *firewall* que protegiesen el tráfico local del exterior. Con estas medidas se trataba de evitar también el riesgo de que un dispositivo comprometido pudiese infectar otros de la misma red de dispositivos conectados a un servidor por proxificación (Fig. 3.1-18). Sin embargo, estas medidas las consideramos insuficientes y propiciaron el que se desarrollaran los contextos de seguridad manejados por el sistema operativo.

Los dispositivos IoT que emplean comunicaciones proxificadas trabajan con *socket* locales que deben ser protegidos: los *socket* de aplicación, las proxificaciones directas y las proxificaciones reversas. Para ello se requiere implementar una protección local complementaria para estos *socket*: los contextos de seguridad. Un contexto de seguridad protegiendo un *socket* local impide conexiones externas con esos *socket* protegidos y restringe las internas a determinados usuarios locales únicamente. De ese modo, este tipo de comunicaciones locales queda controlado y protegido, tal y como se aprecia en la Fig. 3.2-7.

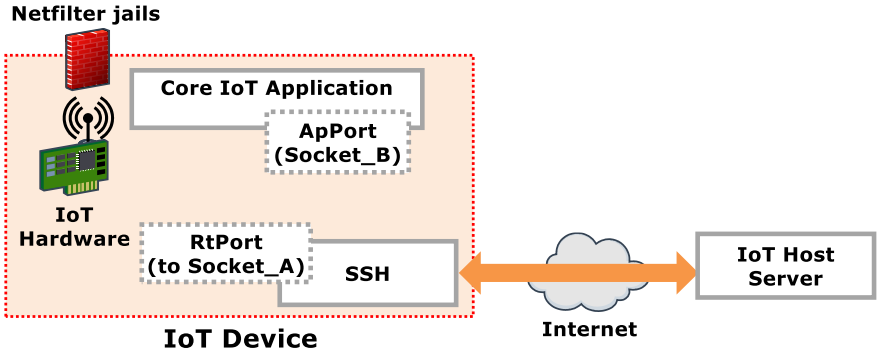


Fig. 3.2-7: Conceptualización gráfica de los contextos de seguridad como Netfilter jails.

La comunicación entre los procesos mediante proxificación entre dos dispositivos consta de tres segmentos (Fig. 3.2-8): el segmento intermedio (1), que comunica ambos dispositivos y queda protegido por la sesión de comunicación entre el agente y el servidor *proxy-firewall*; y los dos segmentos locales (2 y 3), que comunican cada extremo de la proxificación con los procesos respectivos. Cada uno de estos dos segmentos será vulnerable si no se aplica un contexto de seguridad, puesto que cualquier proceso local puede acceder a los *socket* que estén en escucha, lo cual resulta inseguro ya que las aplicaciones que se comunican delegaron toda responsabilidad de seguridad en el agente y el servidor *proxy-firewall*. Por ese motivo, IoTsafe define los conceptos de *socket* protegido y contexto de seguridad.

La necesidad de los contextos de seguridad cobra especial relevancia en el servidor (Device_A), debido a la convivencia de un número elevado de conexiones y proxificaciones establecidas (Fig. 3.2-9). Este hecho hace necesario el control y protección de dichas proxificaciones para evitar que cualquier usuario de Linux local pueda conectarse a alguno de esos *socket* protegidos. Cabe recordar que esos *socket* son ofrecidos con protocolos sin seguridad y normalmente incluso sin autenticación.

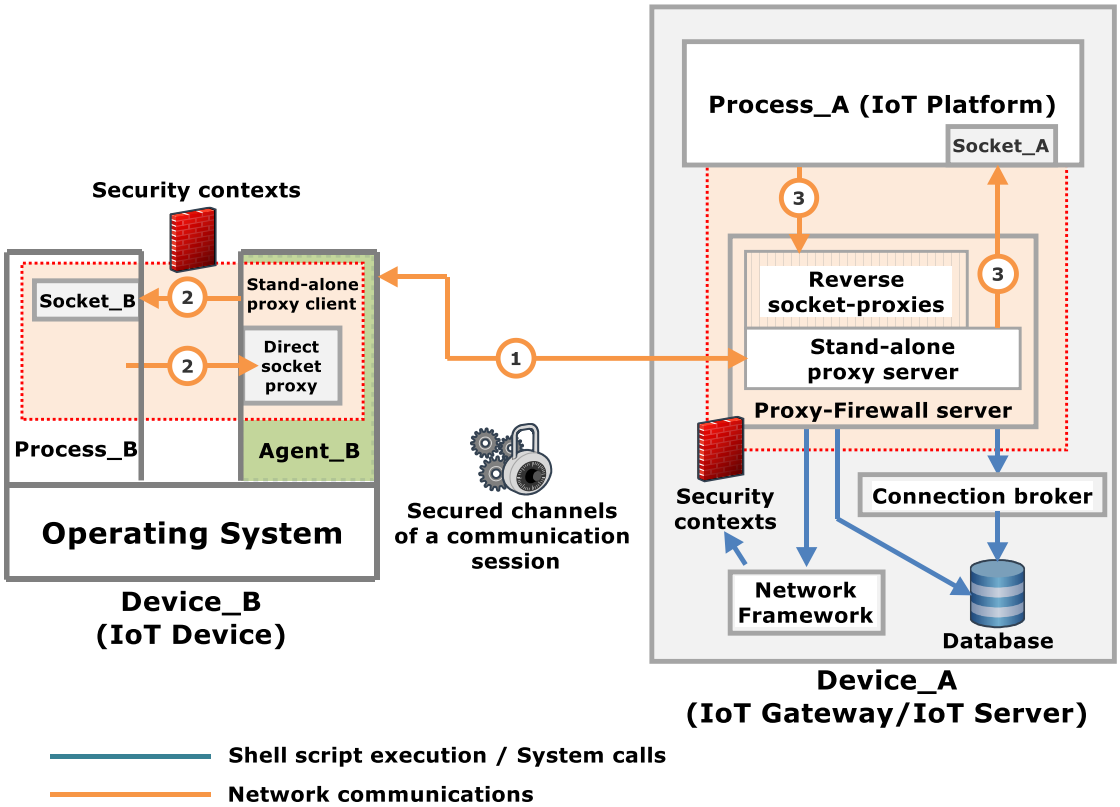


Fig. 3.2-8: ***Esquema de comunicación de IoTsafe en el que los segmentos locales de las mismas son protegidos por contextos de seguridad. En el ejemplo se presentan dos comunicaciones independientes bidireccionales entre Process_B y Process_A, a través de una proxificación directa y reversa respectivamente.***

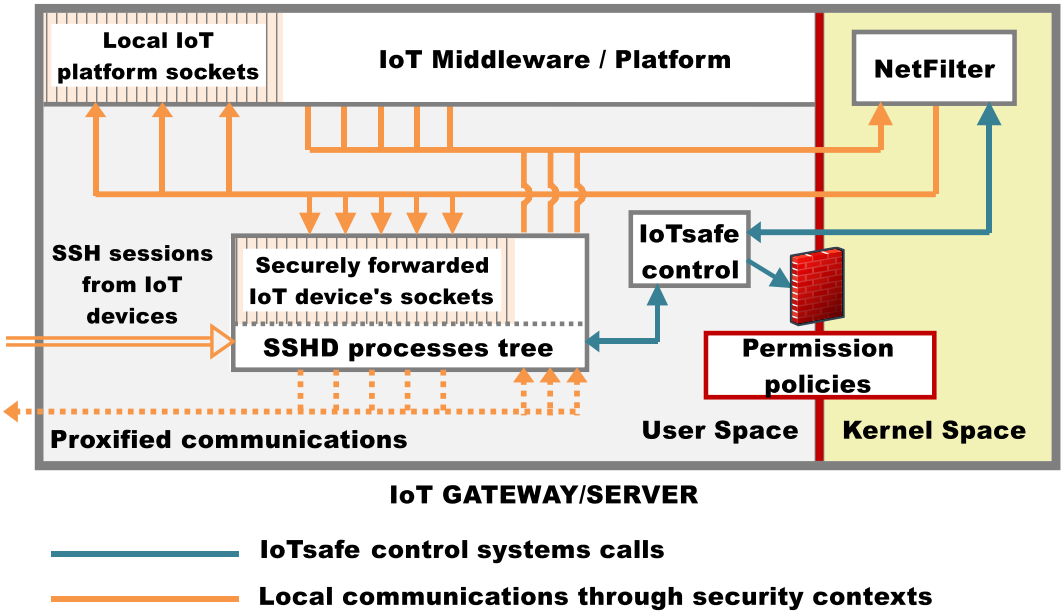


Fig. 3.2-9: ***Diagrama conceptual simplificado de un Gateway/Server de IoTsafe en el que los contextos de seguridad son gobernados por reglas de marcado y filtrado a partir de políticas de permisos controladas por IoTsafe.***

El acceso a los *socket protegidos* está reservado únicamente a aquellos procesos de los usuarios locales Linux que tengan permiso para ello. De este modo, al aplicarse el permiso se establece un contexto de seguridad entre los procesos que ejecute el usuario Linux titular del permiso y el *socket* protegido. El contexto de seguridad (estricto) se define por tanto con el *socket* destino al que una comunicación quiere acceder, y el id de usuario origen titular del permiso cuyo proceso pretende iniciar la comunicación hacia dicho socket.

Los contextos de seguridad se aplican controlando el tráfico local mediante reglas del *framework* de red, que pueden ser de dos tipos: reglas de marcado y reglas de filtrado. Las reglas, a su vez, pueden ser prioritarias o generales. Estas últimas son ejecutadas por defecto en ausencia de una prioritara. La combinación de estas reglas permite proteger las comunicaciones locales entre los *socket* protegidos que utilizan los procesos como la Aplicación IoT (Process_B), el agente del servidor *proxy-firewall* (Agent_B), el "*Proxy-Firewall server Fork*" y la plataforma IoT (Porcess_A) de la Fig. 3.2-10. En esa figura se muestra una implementación de IoTsafe que utiliza SSH como protocolo de proxificación y en el que existen simultáneamente dos comunicaciones independientes:

- La comunicación de la aplicación IoT con la proxificación directa RtPort.
- La comunicación de la plataforma IoT con la proxificación reversa FwPort.

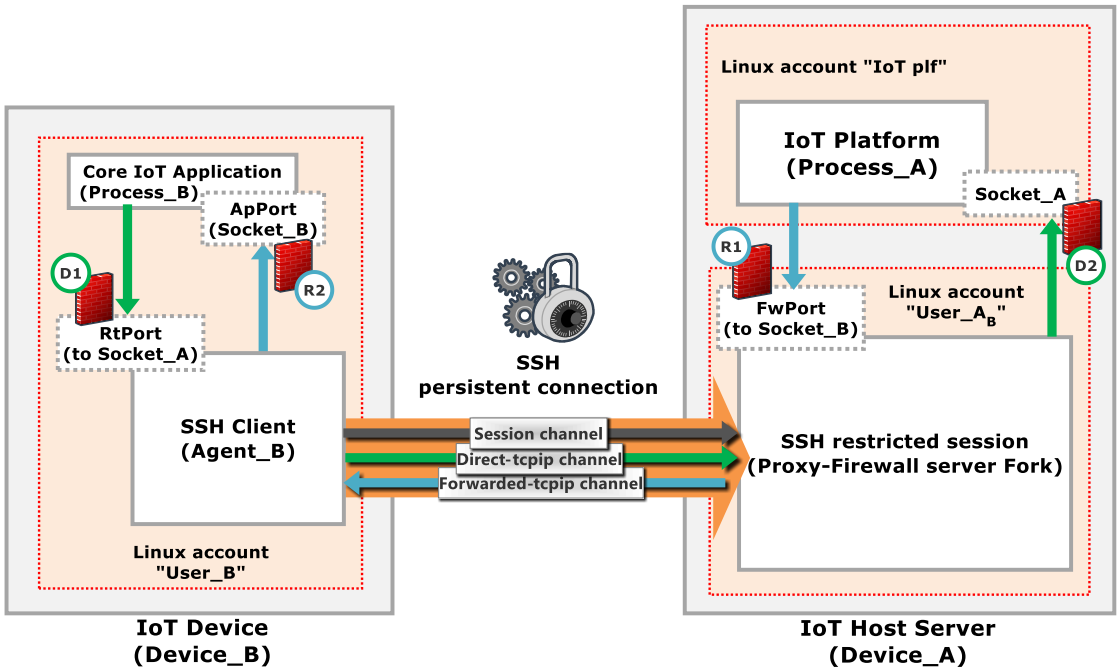


Fig. 3.2-10: Conceptualización gráfica de las comunicaciones a través de los contextos de seguridad en distintos casos junto con los canales SSH empleados. El ejemplo muestra dos comunicaciones independientes entre Process_B y Process_A utilizando una proxificación directa y reversa respectivamente.

Es importante destacar que ambas comunicaciones son independientes. Esto quiere decir que la implementación del sistema es totalmente factible con sólo una de ellas, ya que son bidireccionales. La diferencia entre ambas comunicaciones únicamente radica en la entidad que la inicia. Si la aplicación IoT desea comunicarse de forma asíncrona con la plataforma IoT, necesita disponer de una proxificación directa (RtPort) por la que poder alcanzar el *socket* que proporciona el servicio en la dicha plataforma (Socket_A). Si por el contrario es la plataforma la que requieree comunicarse de forma asíncrona con la aplicación IoT del dispositivo, ésta requiere de una proxificación reversa (FwPort) que la permita acceder al ApPort (Socket_B*)* del dispositivo IoT que le ofrece el servicio.

La definición de las reglas de marcado y filtrado que configuran los *socket* protegidos y los contextos de seguridad utiliza dos elementos: el *socket* local destino de la comunicación, y el usuario cuyo proceso es origen de la comunicación de los paquetes de red. Este último se debe insertar en el tráfico de red mediante el marcado de los paquetes ya que, en la mayoría de los protocolos de red, no existe una traslación de este dato (el usuario origen de la comunicación) al mismo. Por consiguiente, es necesario utilizar un *framework* de red configurado por sistema operativo, para que introduzca ese dato en forma de marcado. Por otro lado, se debe aclarar que el intentar identificar el origen de la comunicación por el *socket* origen resulta generalmente complicado, debido a que las aplicaciones utilizan un *socket* diferente cada vez que solicitan establecer una comunicación. En las comunicaciones descritas en la Fig. 3.2-10 existen cuatro contextos de seguridad, que se analizan a continación.

- **D1**: Contexto de seguridad que habilita al User_B a acceder al RtPort. Como la aplicación del dispositivo IoT la ejecuta el User_B, se puede establecer una comunicación con el *socket* protegido RtPort en cualquier momento.
- **D2**: Contexto de seguridad que habilita al User_A_B para acceder al Socket_A. Cuando el cliente SSH del dispositivo IoT se registró en el servidor IoT, el servidor SSH creó un *fork* de sí mismo en la cuenta User_A_B, que se corresponde con las credenciales usadas por el dispositivo IoT (cada dispositivo IoT tiene una cuenta de usuario Linux distinta en el servidor IoT). El *fork* del servidor SSH se está ejecutando en la cuenta User_A_B y, por tanto, tiene acceso al *socket* protegido Socket_A.
- **R1**: Contexto de seguridad que habilita a la cuenta de usuario "IoT plf" a acceder al FwPort. Dado que en este ejemplo la plataforma se ejecuta con la cuenta de usuario IoT plf, dicha plataforma tiene acceso al *socket* protegido FwPort.
- **R2**: Contexto de seguridad que habilita a la cuenta "User_B" del dispositivo IoT para acceder al ApPort.

Los cuatro contextos de seguridad presentan algunas diferencias entre sí. Mientras en el servidor, los contextos R1 y D2 se aplican a usuarios distintos para acceder a diferentes *socket* protegidos, en el dispositivo IoT los contextos D1 y R2 se aplican al mismo usuario para acceder a *socket* distintos. Esto es debido a que en el dispositivo IoT, pese a ejecutarse el cliente SSH y la aplicación IoT por un mismo usuario Linux, las aplicaciones no se podrían comunicar entre sí de forma arbitraria, al utilizar direcciones locales protegidas: los flujos tienen un mismo origen (cuenta de usuario User_B) pero distinto destino (ApPort y RtPort). En consecuencia, se necesita de un contexto de seguridad diferente para permitir el flujo de cada comunicación. El contexto R2 sería {User_B, ApPort} y el contexto D1 sería {User_B, RtPort}.

Estos contextos son un tipo de *sandbox-jail* que impide que se procesen las comunicaciones con *socket* ajenos a dicho contexto de seguridad. Además, también impide que las comunicaciones dentro de ese "*sandbox-jail*" fluyan de forma distinta a la prevista. Por ejemplo, en el IoT *Host Server,* un contexto de seguridad estricto autoriza que el User_A_B acceda al Socket_A, pero no se permite que del Socket_A se inicien comunicaciones hacia cualquier *socket*. Además, el Usuario_A_B no podrá iniciar comunicaciones hacia otros *socket* protegidos distintos del Socket_A, salvo que tenga permisos extra para nuevos contextos de seguridad.

3.2.2.1. Proceso de marcado y filtrado en los contextos de seguridad

El funcionamiento de los contextos de seguridad requiere primero entender el concepto de "protección del pool de direcciones reservadas" explicado en detalle en el Anexo A. Este procedimiento define tres reglas generales que se aplican a todas las comunicaciones con direcciones IP de origen o destino pertenecientes a dicho pool. Con estas reglas, esas

direcciones IP se protegen y cualquier *socket* que se genere en cualquier programa que utilice esas IPs queda automáticamente protegido.

En la Fig. 3.2-11 se describe en detalle el proceso seguido por los paquetes relativos a una petición cursada bajo un contexto de seguridad. Si se desea habilitar acceso a un *socket* protegido, se debe conceder permiso a la cuenta de usuario que ejecuta el proceso correspondiente. El acceso consiste en la aplicación de una regla prioritaria de marcado que sustituye a la regla general de marcado número (2) y que permite evitar la regla general de filtrado número (3).

A este respecto, cabe destacar el funcionamiento particular de Netfilter (Fig. 2.1-4). Este *framework* de red incluye diferentes tablas de reglas que aplicar sobre el tráfico. Si durante el procesamiento del paquete en cada una de las tablas, el paquete es objeto de alguna regla, se ejecuta la operación definida por esa regla, y el paquete sigue su curso; pero ya no se comprueban más reglas de esa tabla para ese paquete. De ese modo, definir una regla prioritaria sobre una general consiste en un simple aspecto de colocación de la regla en la tabla: la general debe estar en última posición.

Los paquetes dirigidos hacia una dirección protegida se descartan automáticamente si no provienen de otra dirección local protegida (1). Además, IoTsafe define que todo paquete que surja de un *socket* del *pool* de direcciones protegidas deba ser marcado a cero por defecto (2). Si la etiqueta para marcaje IoTsafe es el campo ToS, los valores de este campo deben ponerse a cero. En el caso en que un programa introduzca un marcado en el espacio usuario, este marcado se pierde al llegar al espacio del *kernel,* en donde se aplica esta regla general de marcado. Por tanto, un marcaje a cero condena al paquete a ser descartado en el paso (4). Únicamente, si el tráfico que surje de este *socket* origen tiene un acceso concedido a su destino (un *socket* protegido), entonces Netfilter detectará que dicho paquete debe ser afectado por una

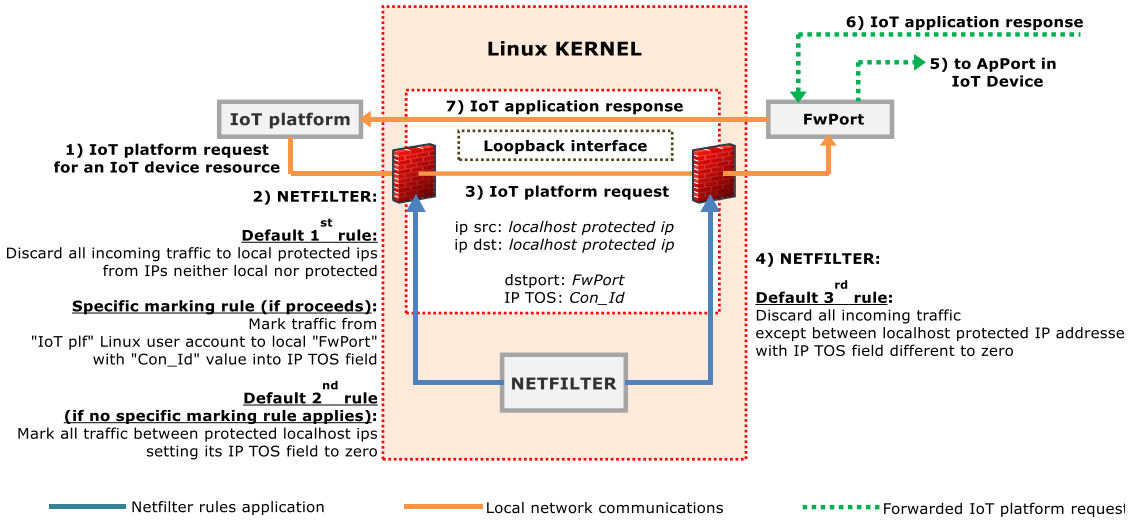


Fig. 3.2-11: Proceso seguido por los paquetes de una petición cursada a través de un contexto de seguridad en el servidor IoT. En este caso, la plataforma IoT requiere de un servicio de la aplicación del dispositivo IoT a la que accede localmente por medio de la proxificación reversa FwPort.

regla de marcado diferente a la general (que lo marcaba a cero) y le marcará en consecuencia con el valor Con_Id que corresponda (3). Este valor Con_Id hace referencia al Id de permiso concedido a un usuario Linux para acceder a un *socket* protegido concreto, y es distinto para cada usuario con permiso para acceder a dicho socket protegido.

El paquete marcado con valor distinto de cero circula localmente y, antes de llegar a su destino, debe volver a pasar por Netfilter. Al considerarse un paquete que ingresa a un *socket* protegido, debe ser comprobado nuevamente (4). En particular, los paquetes de ingreso son descartados automáticamente si provienen de una dirección de red no protegida o si no están marcados. Si el paquete viene de una dirección protegida y está marcado (independientemente de cuál sea el valor marcado), entonces puede ser entregado al *socket* protegido destino. En el caso particular del ejemplo de la Fig. 3.2-11, el *socket* protegido es una proxificación reversa (FwPort) que recibe la petición de la plataforma IoT y la reenvía hacia el *socket* de la aplicación (ApPort) del dispositivo IoT remoto (5).

Una vez la aplicación del dispositivo IoT procese la petición, esta puede emitir una respuesta. En caso de producirse, los paquetes relativos a la misma llegan a la proxificación reversa (FwPort) desde el dispositivo remoto IoT (6) y de ahí circulan localmente (7) hasta volver a la plataforma IoT. Este proceso es automático, al funcionar Netfilter como *stateful firewall*: las respuestas a peticiones permitidas son permitidas sin necesidad de reglas nuevas.

En lo relativo al marcaje y al tratamiento de los paquetes en la fase de ingreso al socket protegido (4), IoTsafe no verifica la procedencia de dicho paquete en base a su marcado, es decir, el esquema de comunicación planteado hace que no sea necesario verificar en el paso (4) si el número del campo ToS (Con_Id) corresponde al de un usuario local con un permiso válido para acceder al *socket* protegido en cuestión. Esto permite acelerar el flujo de paquetes cursados, al no requerir de ninguna lógica compleja extra. Sólo se ha de comprobar que proviene de una dirección del pool de direcciones locales protegidas, y que el marcado es distinto a cero.

El número marcado en el campo ToS, sin embargo, debe ser distinto para cada usuario con acceso concedido a un mismo *socket* protegido. Esto es un requisito para poder identificar fácilmente las reglas de marcado activas relativas a cada usuario; en definitiva, para diferenciar los contextos de seguridad que aplican en ese socket protegido en particular. En caso de tener que suprimir la aplicación de uno de los contextos, el marcado del campo ToS (Con_Id) junto con el *socket* destino, sirven de identificador unívoco del contexto de seguridad en esa máquina. Con esos datos, el connection-broker de IoTsafe puede consultar en la base de datos la cuenta de usuario relativa a cada contexto de seguridad. El valor diferente de Con_Id permite facilitar la identificación de los flujos de datos a posibles herramientas AI que operen monitorizando el tráfico de forma pasiva y que necesiten clasificar en tiempo real la procedencia de los flujos de datos. En este escenario, la vulnerabilidad de las comunicaciones queda restringida a la

posibilidad de que un usuario o proceso pueda obtener privilegios elevados que le permitan controlar el *framework* de red.

A continuación, se resumen las características principales de los contextos de seguridad y de los elementos que interactúan con ellos (los permisos, los *socket* protegidos, las comunicaciones proxificadas y el *framework* de red):

- Un *socket* protegido normalmente está aislado por el *framework* de red (Netfilter). Ninguna comunicación puede acceder a él y tampoco puede generar comunicaciones hacia otros destinos.
- Los *socket* protegidos no se protegen con reglas específicas. Existen reglas generales del *framework* de red, aplicables en el inicio del dispositivo, que protegen rangos de direcciones locales para que cualquier *socket* que utilice esa dirección de red local sea automáticamente un *socket* protegido. Esto consigue que, con un número muy reducido de reglas, se protejan todos los *socket* necesarios (Anexo A).
- Un *socket* protegido nunca es accesible remotamente. Idealmente, sólo debe ser accesible desde direcciones locales protegidas.
- Para que un *socket* protegido pueda ser accesible localmente, debe existir un permiso (estricto o amplio) que habilite a un usuario local del sistema operativo para que sus procesos puedan iniciar comunicaciones hacia ese *socket* protegido.
- Una vez un usuario del sistema operativo ha conseguido un permiso habilitante, las comunicaciones hacia ese *socket* protegido son posibles y las respuestas de ese *socket* son recibidas por el programa origen de las comunicaciones. NetFilter opera como un *firewall stateful* y permite automáticamente dichas respuestas.
- Un permiso estricto habilita un número indeterminado de conexiones locales con origen en cualquier proceso de un usuario local y destino ese *socket* protegidom en particular al que hace referencia dicho permiso.
- Un permiso amplio habilita a cualquier usuario perteneciente al grupo de usuarios Linux al que hace referencia dicho permiso a acceder al *socket* protegido.
- Los permisos son configuraciones que se traducen a reglas de marcado que el *framework* de red del sistema operativo utiliza para diferenciar el tráfico local y permitir ese tráfico con el *socket* protegido.
- El marcado del tráfico local se puede hacer de diversas maneras, pero por sencillez, IoTsafe sugiere que se utilice el campo ToS de IPv4, afectando únicamente en los paquetes de comunicación con origen o destino un *socket* protegido (Anexo B). Cabe destacar que el tráfico marcado es estrictamente local e indepediente del protocolo de aplicación que delegue en IoTsafe las funcionalidades de seguridad.
- Cada permiso se traduce en una única regla de marcado. La supresión de dicha regla del *framework* de red del sistema operativo anula el efecto del permiso respectivo.
- El sistema que emplea IoTsafe se reserva el uso exclusivo del *framework* de red en las

comunicaciones locales que involucren direcciones pertenecientes al pool de direcciones protegidas. El marcado de paquetes desde el espacio usuario en el campo ToS siempre es sobreescrito por el *framework* de red en estas circunstancias, para para ponerlo a cero (no marcado) si no tiene permiso habilitante; o a un valor distinto de cero si sí lo tiene. Este valor es el Con_Id y sirve de identificación del flujo permitido hacia el *socket* protegido.

- Los valores que Con_Id puede tomar son limitados en el caso en que se utilice el campo ToS: de 1 a 255. Por consiguiente, si se implementa IoTsafe con este tipo de marcado sólo se permiten 255 permisos simultáneos a un *socket* protegido específico.
- El número de permisos que requiere un sistema Linux para acceder a un *socket* protegido no depende del número de procesos que necesitan de dicho acceso para establecer comunicaciones, sino del número de cuentas de usuario Linux diferentes necesarias para que se ejecuten esos procesos. A continuación, se listan algunos ejemplos:
    - Si cada proceso se ejecuta desde una cuenta de usuario Linux diferente, el número total de permisos estrictos que requiere el sistema es el número de procesos totales que requieren la comunicación.
    - Si todos los procesos que requieren acceder a un *socket* protegido se ejecutan desde la misma cuenta de usuario Linux, sólo se requiere un único permiso estricto.
    - Si existen procesos en distintas cuentas de usuario que requieren establecer comunicación con un recurso protegido, se puede optar por obtener un permiso estricto para cada cuenta de usuario, o por formar un grupo de usuarios Linux con esas cuentas de usuario y conceder un único permiso amplio a dicho grupo de usuarios.
- Normalmente, los procesos que requieren comunicarse con un *socket* protegido se ejecutan empleando una dirección IP protegida para que los *socket* empleados sean también protegidos. Este hecho no requiere de permisos especiales extra ya que Netfilter es un *stateful firewall* y permite que los *sockets* de los programas origen de las comunicaciones reciban las respuestas de los *socket* protegidos destino de dichas comunicaciones. Esta medida, además protege los *socket* origen de comunicación unos de otros: si múltiples procesos ejecutados bajo una cuenta con permiso para acceder a un *socket* protegido tratan de comunicarse entre sí, no podrán, pese a estar bajo un contexto de seguridad ya que el contexto habilita comunicación únicamente hacia el *socket* protegido objeto de dicho permiso.
- Las reglas de marcado y filtrado empleadas por IoTsafe no requieren ninguna personalización del *kernel*, son estándar y reconocidas por el *framework* de red Netfilter que Linux implementa por defecto y además son configurables con la herramienta iptables.

- Las comunicaciones proxificadas tienen un carácter orientado a conexión y los permisos y contextos de seguridad se establecen en el tiempo de forma duradera para acelerar posteriormente cualquier comunicación puntual entre el dispositivo IoT y el servidor. Esto evita tener que repetir los procesos inherentes al establecimiento de las proxificaciones (sesiones SSH) y de los contextos de seguridad.

3.2.3. **IoTsafe como middleware entre la aplicación y las comunicaciones**

Esta sección pretende dar respuesta a la cuestión C5 planteada en la sección 1.3:

C5: ¿Cuál sería el esquema de seguridad técnico propuesto que habilite una comunicación horizontal segura?

El diseño del método de manejo de las comunicaciones proxificadas en el servidor, presentado anteriormente, ofrece múltiples posibilidades a empresas y cuerpos reguladores en el campo de la seguridad de las comunicaciones IoT, pero también presenta nuevas oportunidades en el desarrollo de *software* y servicios nuevos de manera más eficiente. El esquema de contextos de seguridad propuesto por IoTsafe se basa en políticas de desacoplamiento de seguridad utilizando como interfaz la API de *socket*. Este enfoque puede mejorar la interoperabilidad y ayudar al cumplimiento de los procedimientos de certificación, ya que proporciona una “horizontalización de los *verticals* de IoT” [90].

El *software* que integra IoTsafe en los dispositivos y opera como *middleware*, debería ser el único del dispositivo IoT capaz de comunicarse externamente, proporcionando así una proxificación transparente entre los *socket* de comunicación de la aplicación del dispositivo IoT y los que están en el *gateway*/servidor IoT. Las comunicaciones proxificadas que ocurren localmente entre procesos en el mismo dispositivo, ya sea en el dispositivo IoT o en el gateway/servidor, están protegidas localmente a través de los contextos de seguridad antes mencionados.

Por otro lado, las comunicaciones locales entre *socket* dentro del servidor (Fig. 3.2-5) se conmutan de manera segura de un *socket* a otro, siguiendo un enfoque de conmutación de circuito virtual. Esta comunicación local dentro de *gateway*/servidor IoT está también diseñada y protegida de manera transparente utilizando los contextos de seguridad, que a su vez emplean únicamente funcionalidades estándar del *kernel* de Linux. El sistema operativo se presenta como un "*hipervisor de comunicación*" transparente a las comunicaciones, de un modo similar a como los *hipervisors* tradicionales manejan máquinas virtuales. Tal enfoque de desacoplamiento de seguridad contribuye a reforzar las siguientes características:

- **Desarrollo eficiente de aplicaciones**: los problemas de seguridad de la comunicación están fuera del alcance de la aplicación del dispositivo IoT y pueden delegarse en un *software* complementario de seguridad que integra IoTsafe y que puede ser mantenido por terceros (profesionales especializados en seguridad).

- **Mayor simplicidad en las comunicaciones de aplicación**: los datos transmitidos extremo a extremo son generados por aplicaciones no seguras a través de un canal seguro, basado en comunicacones proxificadas que resulta transparente.
- **Comunicación independiente de la tecnología**: las fuentes de datos del dispositivo IoT proporcionados a través de los *socket* de red locales del propio dispositivo pueden transmitirse de forma segura, transparente e independiente a través de la tecnología de comunicación subyacente, mediante el uso de técnicas de desacoplamiento de seguridad. Por lo tanto, los recursos locales del dispositivo IoT se pueden reenviar como simples *socket* de red hacia un *gateway*/servidor IoT. Los *socket* pueden considerarse una interfaz mucho más sencilla que la utilizada por la mayoría de las APIs empleadas por otros middleware de seguridad y, a su vez, los recursos remotos (de las plataformas IoT, por ejemplo) también se pueden reenviar de forma segura al dispositivo IoT.

Como resultado, todos los aspectos de seguridad pueden delegarse casi por completo en la capa de comunicación proxificada provista por IoTsafe. Estas configuraciones y módulos de seguridad pueden ser actualizados y mantenidos por empresas externas especializadas en seguridad, que ofrecerían un servicio análogo al proporcionado actualmente por las firmas de seguridad antivirus para computadoras y servidores. Esto podría permitir un procedimiento de prueba de "w*hite-box*" que permitiría el cumplimiento de los niveles más altos de certificación de seguridad de manera más asequible durante todo el ciclo de vida del dispositivo.

Simultáneamente, la interoperabilidad también se simplificaría, ya que las interfaces de capa de comunicación relacionadas consisten en *socket* locales simples, manejados de manera transparente a través de contextos de seguridad: las aplicaciones del servidor IoT perciben todos los recursos de los dispositivos IoT como entidades locales (*socket* simples dentro de la máquina) y también ocurre lo mismo con los dispositivos IoT, que perciben como locales los proporcionados por la plataforma / *middleware* IoT de gateway/servidor IoT.

La Fig. 3.2-12 muestra un ejemplo de esta horizontalización, en el que se incluyen múltiples tipos de dispositivos IoT: Industriales, de recursos limitados y de alto rendimiento. En este ejemplo, todos los dispositivos IoT proxifican los *socket* de las aplicaciones locales IoT hacia el *IoT Host*, en donde:

- Se pueden establecer comunicaciones entre dichos *socket* (M2M).
- Las plataformas pueden conectarse a dichos *socket* y acceder a los recursos de los dispositivos IoT si cuentan con el permiso de acceso correspondiente. Esto significa que la conexión de una plataforma con un dispositivo IoT no manejado inicialmente por dicha plataforma es totalmente factible, haciendo falta únicamente disponer del permiso de acceso correspondiente.
- Este esquema permite comunicaciones entre dispositivos *IoT Host* diferentes y habilita que plataformas IoT remotas puedan acceder a dispositivos IoT de un host específico.

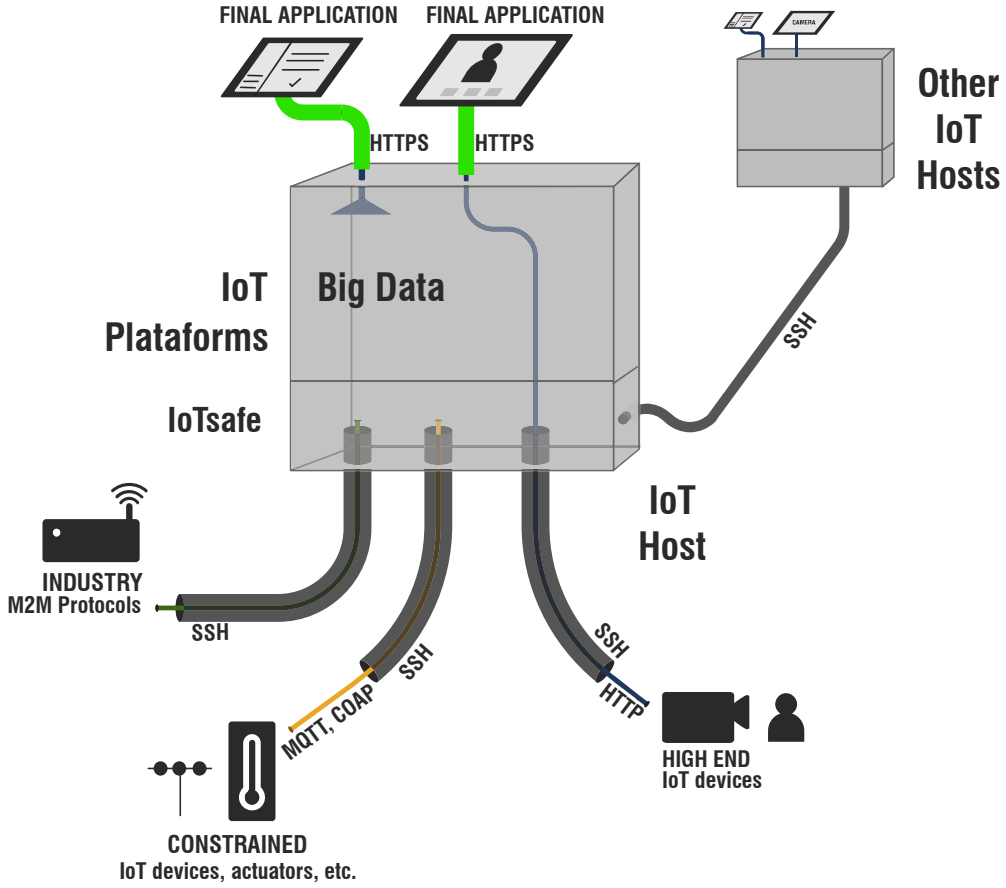


Fig. 3.2-12: Ejemplo conceptual de horizontalización de verticals. Múltiples dispositivos IoT comparten infraestructura común de conexión: el espacio IoTsafe compuesto por proxificaciones en sockets seguros.

El espacio existente en el servidor compuesto por todas las proxificaciones locales y su tráfico local, libre de protocolos de seguridad, facilita por tanto la integración de proyectos IoT diferentes al simplificarse notablemente el *middleware* necesario:

- Las comunicaciones no requieren autenticación ni protocolos de seguridad.
- Los permisos de acceso a cada proxificación se gestionan de la misma manera: a través del sistema operativo y por contextos de seguridad.
- La gestión de las comunicaciones es transparente a los programas de los dispositivos IoT e independiente de las comunicaciones entre dichos dispositivos y las plataformas IoT del servidor. Los cambios en los módulos de seguridad del servidor y del dispositivo IoT son independientes de los binarios de los dispositivos IoT y del *software* que conforme las plataformas IoT.
- La IA necesaria para detectar posibles ataques o vulnerabilidades no descubiertas se simplifica notablemente al poder caracterizar más fácilmente el tráfico entre los dispositivos IoT y las plataformas: Este tráfico se cursa localmente entre proxificaciones locales del servidor y puede analizarse con técnicas *Deep Packet Inspection* (DPI) de forma más sencilla, al ser comunicaciones sin ningún tipo de protocolo de seguridad.

Se confirma por tanto que la robustez y seguridad de un esquema no necesariamente implica una gran complejidad. Diseños muy elaborados con numerosos módulos *software* pueden resultar más complejos de analizar y monitorizar que diseños más sencillos: un ataque exitoso en esas circunstancias puede pasar desapercibido más fácilmente y, por tanto, se requeriría más tiempo para implementar la solución correspondiente.

El método y sistema completo queda detallado en el Anexo B. El Anexo C incluye además distintos ejemplos de implementación de IoTsafe.

3.3. DESCRIPCIÓN DETALLADA DEL ESQUEMA DE DESACOPLE DE SEGURIDAD EN LAS COMUNICACIONES

En esta sección se presenta formalmente una posible realización del esquema de seguridad que emplea IoTsafe, que sirve como respuesta a la cuestión C6 planteada en la sección 1.3:

C6: ¿Es posible diseñar un sistema de seguridad con las características anteriores que además permita la integración de otras tecnologías software IoT complementarias?

Un sistema que implemente un esquema de comunicación como el propuesto por IoTsafe debería permitir una sencilla integración de otras herramientas de seguridad que cubran aspectos complementarios. Un ejemplo claro de estas tecnologías complementarias podrían ser CONCORDIA[87], RERUM[88] o tecnologías *Machine Learning* (ML)/*Deep Learning* (DL) para la mitigación de ataques contra dispositivos y servicios IoT.

En la Fig. 3.3-1 se describe una implementación de IoTsafe basada en el desacoplo de seguridad, y en el uso de contextos de seguridad que, además de implementar las características de diseño explicadas en apartados anteriores, permite demostrar cómo IoTsafe puede acoplarse a otras tecnologías IoT con facilidad. Esta realización está orientada a un ecosistema IoT, y los términos utilizados en esta descripción tienen el significado especificado en el Glosario. El ejemplo de esquema de seguridad propuesto se considera una realización particular de un método y sistema de comunicación más generalista, descrito en el Anexo B. Dicha realización aglutina la mayoría de conceptos y principios de funcionamiento, que han sido explicados por separado y en detalle a lo largo de este capítulo. Por lo tanto, en este ejemplo se describen nuevamente algunas de los conceptos, pero se considera oportuno para poder así presentar de forma más completa un ejemplo generalista de IoTsafe [110].

[87]CONCORDIA ecosystem: Cyber security cOmpeteNCe fOr Research anD InnovAtion, https://www.concordia-h2020.eu/
[88]RERUM: REliable, Resilient and secUre IoT for sMart city applications, https://ict-rerum.eu/

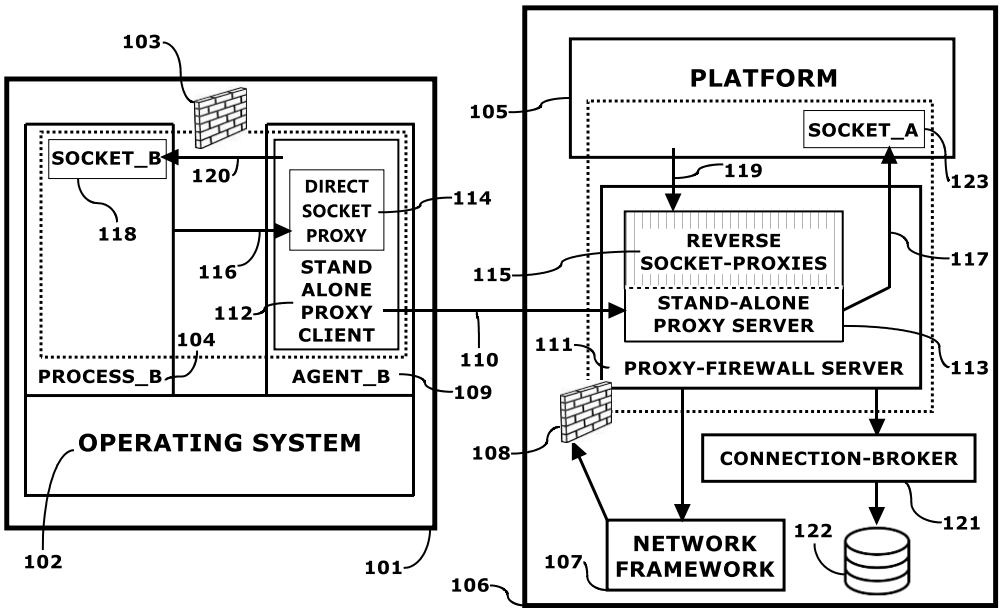


Fig. 3.3-1: Diagrama conceptual de diseño de IoTsafe [110]***.***

En el esquema concreto que se presenta, un ejemplo de dispositivo_B (101) con el que se pueden identificar todos aquellos dispositivos IoT del ecosistema que sean considerados "*edge-devices*" de dicho ecosistema. En este ejemplo, los dispositivos pueden funcionar con *firmware* o sistemas operativos (102) de tipo POSIX. Un ejemplo de este tipo de sistemas operativos son los de la familia Linux o alguna de sus variantes. En todo caso, disponen de un *kernel* que puede hacer uso de un *framework* de red como Netfilter, que puede estar en el *kernel* o ser cargado mediante módulos externos, y a través del cual se ingresan reglas de marcado y filtrado (103). Este dispositivo_B posee al menos un proceso_B (104) que desarrolla la funcionalidad principal del dispositivo y que requiere de comunicación con un *software* de gestión como puede ser una plataforma (105) de un dispositivo_A (106).

En el ecosistema IoT que se presenta, un ejemplo de dispositivo_A correspondería con el host IoT al que se conectan los dispositivos IoT. En este ejemplo, el dispositivo_A cuenta con un sistema operativo también tipo POSIX y también de la familia Linux, pero esto es algo que no resulta imprescindible, ya que al ser sistemas operativos de tipo POSIX, la interoperabilidad entre ellos es factible. En todo caso, el sistema operativo del dispositivo_A, al ser también de la familia Linux en este ejemplo, también cuenta con un *kernel* con *framework* Netfilter (107) mediante el que también se ingresan reglas de marcado y filtrado (108). El sistema operativo del dispositivo_A, al ser utilizado en un host IoT, deberá ser multiusuario y con capacidad de separación de privilegios. De ese modo, cada dispositivo_B que se conecte al mismo podrá utilizar una cuenta de sistema operativo diferente en el dispositivo_A. Esto permite a Netfilter diferenciar los flujos de comunicación y crear contextos de seguridad que extiendan la protección de las comunicaciones en el área local para cada dispositivo.

La seguridad en las comunicaciones entre el dispositivo_A y el dispositivo_B la proporciona un agente_B (109) del dispositivo_B que establece una sesión segura (110) con un servidor *proxy-firewall* (111) del dispositivo_A. Esta sesión segura utiliza un protocolo de seguridad que se ejecuta entre un cliente *proxy stand-alone* (112) y un servidor *proxy stand-alone* (113). El primero forma parte del agente_B, y el segundo forma parte del servidor *proxy-firewall*. La sesión que establecen entre ellos permite realizar proxificaciones entre los dos dispositivos. Un ejemplo de este tipo de protocolos es SSH, el cual dispone de implementaciones que funcionan como cliente *proxy stand-alone* y servidor *proxy stand-alone* como las requeridas en este ejemplo. Algunos ejemplos de estas implementaciones son OpenSSH[89] y Dropbear[90].

El agente_B, utiliza la sesión establecida para proxificar conexiones entre ambos dispositivos según esté especificado en el perfil_B_A que utilice dicho agente para establecer la sesión. De este modo, y según la información contenida en dicho perfil, el agente_B crea los *socket-proxies* directos (114) en el dispositivo_B y *socket-proxies* reversos (115) en el dispositivo_A que correspondan. El establecimiento de proxificaciones, por tanto, puede realizarse en ambos sentidos, aunque la sesión entre el agente y el servidor *proxy-firewall* la inicie el agente_B.

Una vez establecidas las proxificaciones, tanto el proceso_B como la plataforma IoT pueden comunicarse utilizando los *socket-proxies*. Si la comunicación la inicia el proceso_B, utilizará en este ejemplo de comunicación el *socket-proxy* directo local (114) que genera el agente_B para comunicarse con el *socket_A* (123). Dicha comunicación segura tiene tres partes:

- La primera (116), es un segmento local que comprende la comunicación entre el proceso_B y el *socket-proxy* directo del agente_B. Este segmento de comunicación queda protegido por el contexto de seguridad estricto configurado por el agente_B al arrancar el sistema operativo del dispositivo_B por medio de Netfilter.
- La segunda, entre el dispositivo_B y el dispositivo_A, está cifrada por la sesión SSH (110).
- La tercera (117), entre el servidor *proxy-firewall* y el *socket_A* de la plataforma queda protegida por un contexto de seguridad amplio configurado por el servidor *proxy-firewall* a través del Netfilter (107) del dispositivo_A.

Cuando la plataforma IoT requiere iniciar una comunicación con el *socket_B* (118) del proceso del dispositivo_B entonces utiliza uno de los *socket-proxies* reversos (115) generados en el dispositivo_A; en este ejemplo de comunicación serían el correspondiente al alias del *socket_B* y al id_B del dispositivo_B (id_DispSocket) al que pretende acceder la plataforma IoT. Esta comunicación también consta de tres partes: la primera (119), entre la plataforma IoT y el *socket-proxy* reverso, queda protegida por un contexto de seguridad estricto configurado por

[89] "OpenSSH" https://www.openssh.com/
[90] "DropBear" https://matt.ucc.asn.au/dropbear/dropbear.html

Netfilter en el dispositivo_A. La segunda, entre el dispositivo_B y el dispositivo_A se encuentra protegida por la sesión segura (110) y finalmente la tercera (120), entre el agente y el *socket_B* (118) del proceso_B del dispositivo_B, se encuentra protegida por otro contexto de seguridad estricto del dispositivo_B.

La gestión de todas las sesiones establecidas, *socket-proxies* asignados y permisos de acceso a los mismos para usuarios del sistema operativo del dispositivo_A, son manejados por el *connection-broker* (121) y su base de datos (122) a instancias del servidor *proxy-firewall*.

El acceso a los *socket_B* en el dispositivo_B o a los *sockets* protegidos del dispositivo_A se facilita mediante unas reglas de marcado prioritarias que definen los respectivos contextos de seguridad estrictos. Las reglas de marcado en este ejemplo se aplican incluyendo el con_Id en el campo ToS del protocolo IP. El marcado de tráfico siguiendo esta estrategia facilita la operación de Netfilter y la implementación del servidor *proxy-firewall*, pero limita a 255 el número de contextos de seguridad distintos aplicables a cada *socket* protegido. Esta cifra, en el caso de los *socket-proxies* reversos, es suficiente debido a que los dispositivos IoT, suelen ser dispositivos de recursos limitados que no suelen estar preparados para permitir múltiples conexiones simultaneas: la información y los servicios que ofrecen se presentan al usuario final o a la aplicación final a través de recursos propios de la plataforma a la que se conectan.

Los *socket_A*, son también *socket* protegidos y también requieren de permisos de acceso para las cuentas de cada dispositivo_B con procesos que requieran conectarse. El número de contextos estrictos de seguridad necesario en este caso por cada *socket_A* sería tan grande como el número de dispositivos_B que requieran comunicación con dicho *socket*, y en la mayoría de los proyectos IoT sí se superan los 255. Por ello, en este caso, se aplica un solo contexto de seguridad amplio a todo un grupo de usuarios del dispositivo_A, de modo que las cuentas de usuario del dispositivo_A que utiliza cada dispositivo_B al conectarse, compartan el permiso_A_G; para ello se incluyen esas cuentas de usuario en un mismo grupo con dicho permiso.

El esquema de comunicación presentado consigue por tanto ofrecer seguridad a las comunicaciones entre los dispositivos y la plataforma IoT a través de proxificaciones transparentes. Estas comunicaciones pueden extenderse más allá de la propia plataforma IoT gracias a los contextos de seguridad (ver Anexo C, ejemplos en secciones E4, E5, E6, E7, E8 y E9) permitiendo, entre otras cosas, que otros procesos locales o remotos del servidor IoT puedan acceder a los *socket* protegidos que los dispositivos IoT tienen en ese servidor. Dichos procesos perciben los dispositivos IoT como elementos locales dentro del servidor en el que corren. Al mismo tiempo, si dichos procesos tuviesen que desplegar servicios en cada dispositivo_B, también podrían hacerlo mediante *socket* protegidos locales (proxificaciones directas) dentro de cada dispositivo_B. El *software* IoT (*process_B*) de los dispositivos_B entonces percibiría a las plataformas IoT (u otros procesos del *host* que son fuente de dichas proxificaciones directas) como entidades locales dentro del dispositivo IoT. Esto se consigue gracias a un espacio virtual

de comunicaciones en el servidor IoT, que consiste en *socket* protegidos operados por IoTsafe a través del *kernel* de Linux, por medio de contextos de seguridad, que consisten en permisos, marcado y filtrado de paquetes de red locales. Esta infraestructura local de comunicación hace por tanto más factible una integración sencilla de otros sistemas de comunicación, al tratar todo el tráfico cursable entre los distintos actores de forma exclusivamente local y sin necesidad de tener que integrar protocolos de seguridad en las aplicaciones involucradas.

3.4. IOTSAFE SEGÚN LAS RECOMENDACIONES DE LA UNIÓN INTERNACIONAL DE TELECOMUNICACIONES (UIT)

La UIT en su recomendación X.200 [117] define las normas de interconexión entre sistemas abiertos[91] (OSI) para el modelo de referencia básico. Este modelo define siete capas determinadas por ciertos principios que persiguen optimizar el sistema resultante en varios campos: funcionalidad, escalabilidad, interoperabilidad, independencia y simplicidad, entre otros. Así mismo, la UIT hace un análisis específico relativo a la seguridad de las comunicaciones en su recomendación X.800 [118], y de igual modo, enumera otra serie de principios que ayudan a determinar la asignación de servicios de seguridad a las capas y la consiguiente ubicación de los mecanismos de seguridad**.** La distribución de los servicios de seguridad en las diferentes capas se presenta en la Tabla I.

TABLA I
RELACIÓN ENTRE LOS SERVICIOS DE SEGURIDAD Y LAS CAPAS OSI [114]

Servicios	Capas						
	1	2	3	4	5	6	7
Autenticación de la entidad par	·	·	S	S	·	·	S
Autenticación del origen de los datos	·	·	S	S	·	·	S
Servicio de control de acceso	·	·	S	S	·	·	S
Confidencialidad en modo con conexión	S	S	S	S	·	S	S
Confidencialidad en modo sin conexión	·	S	S	S	·	S	S
Confidencialidad de campos seleccionados	·	·	·	·	·	S	S
Confidencialidad del tren de datos	S	·	S	·	·	·	S
Integridad en modo con conexión con recuperación	·	·	·	S	·	·	S
Integridad en modo con conexión sin recuperación	·	·	S	S	·	·	S
Integridad en modo con conexión de campos seleccionados	·	·	·	·	·	·	S
Integridad en modo sin conexión	·	·	S	S	·	·	S
Integridad en modo sin conexión de campos seleccionados	·	·	·	·	·	·	S
No repudio. Origen	·	·	·	·	·	·	S
No repudio. Entrega	·	·	·	·	·	·	S

S Sí, el servicio debe incorporarse en las normas de la capa como opción del proveedor.
· No se suministra el servicio.

[91] **Sistema abierto:** Representación dentro del modelo de los aspectos de un conjunto de uno o más computadores, el soporte lógico asociado, periféricos, terminales, operadores humanos, procesos físicos, medios de transferencia de información, etc., que forma un todo autónomo que cumple los requisitos de las normas relativas a OSI en su comunicación con otros sistemas [113].

La capa 7 es la que más servicios de seguridad puede llegar a concentrar, siendo la más compleja. Por este motivo, resulta comprensible que las aplicaciones IoT sean muy sensibles a cualquier cambio necesario en los servicios de seguridad. Esto se debe a que la 7ª capa, al ser la más alta en el modelo de referencia OSI, siempre va a acabar integrada en la aplicación IoT propiamente dicha, ya que en el modelo de referencia no diferencia entre las funcionalidades de comunicación y las funcionalidades propias de la aplicación del dispositivo. Este hecho fuerza a que la complejidad de la seguridad de esta capa quede inherentemente vinculada a la aplicación, lo que podría llegar a entenderse como una vulneración del principio de definición de capas si se llegase a considerar a la aplicación IoT como un "nivel extra" que hace uso de los niveles OSI (la capa de aplicación OSI hace referencia sólo al protocolo de comunicación de la aplicación). En ese caso, las funcionalidades propias de la aplicación IoT y su estrecha vinculación con los servicios de seguridad del nivel 7 contravendrían los principios OSI de definición de niveles en la recomendación X.200, sección 6.2, párrafos c), d), f), g) e i) [117]:

c) *"Conviene crear capas separadas para tratar las funciones que son a todas luces diferentes en el proceso ejecutado o en la tecnología aplicada."* En la pila TCP/IP, la capa de aplicación es tradicionalmente compleja, integrando los niveles de sesión y presentación también. Los aspectos de seguridad de las comunicaciones tienden a complicar en exceso el funcionamiento del sistema, debido a su dependencia con el código propio de la aplicación y este hecho refuerza la idea usada por IoTsafe: desacoplar las funcionalidades de seguridad de la aplicación misma en una nueva capa.

d) *"Se deben reunir las funciones similares en una misma capa."* De nuevo, este punto refuerza la idea de la necesidad de concentrar en una capa las funciones específicas de seguridad, pero que sin incluir necesariamente a la aplicación en sí. Esto podría ser abordable considerando el desacoplo de seguridad propuesto en IoTsafe.

f) *"Se debe crear una capa con funciones fácilmente localizables, de modo que* (cuando proceda) *se pueda diseñar de nuevo* (la capa) *totalmente y modificar profundamente sus protocolos para aprovechar los nuevos adelantos arquitecturales y tecnológicos de los soportes físicos y lógicos sin cambiar los servicios que han de prestar y recibir las capas adyacentes."* IoTsafe persigue esta misma filosofía. La independencia entre la aplicación IoT y la seguridad permite que se diseñen nuevos protocolos de seguridad, sin necesidad de modificar el funcionamiento de las aplicaciones IoT.

g) *"Se debe crear una frontera donde pueda ser útil, en un momento dado, normalizar la interfaz correspondiente."* La "frontera" propuesta por IoTsafe consiste en los "contextos de seguridad" aplicados a *socket* locales de red. Estos dos elementos constituyen la base de una interfaz de seguridad suficientemente flexible e independiente que permite aplicar la capa de seguridad a una gran variedad de aplicaciones IoT de diseño arbitrario.

i) "Se debe permitir la modificación de funciones o protocolos dentro de una capa sin que ello afecte a otras capas." La capa de seguridad que presenta IoTsafe permite precisamente esto último, gracias al desacoplo de seguridad propuesto.

Los principios propuestos en la recomendación X.800 sección 6.1.1 para la asignación de servicios de seguridad [118]**,** también refuerzan la idea de concentrar en una capa los aspectos de seguridad de forma independiente a la aplicación gracias a IoTsafe. Esta propuesta se respalda especialmente en los principios recogidos en los apartados a), d), g) y h) de dicha recomendación**:**

a) *"El número de maneras alternativas de proporcionar un servicio* (de seguridad) *debe reducirse al mínimo."* Una manera eficaz de abordar este propósito consistiría en de concentrar en una capa específica los aspectos de seguridad que afecten de forma directa en el desarrollo de las aplicaciones, para poderlos desacoplar mediante IoTsafe.

d) "*Debe evitarse la violación de la independencia de las capas."* Existen protocolos de seguridad que requieren que las aplicaciones cumplan con determinados requisitos empleados en su diseño, en particular el uso de framework de seguridad de desarrollo específicos. Si entendiésemos a la propia aplicación como una "capa extra" sobre el sistema OSI, la falta de una capa que aglutinase esas funcionalidades de seguridad impediría precisamente esta independencia de las capas. Por ejemplo, el actualizar la seguridad relativa al protocolo de comunicación que utiliza la aplicación podría implicar actualizar toda la aplicación.

g) *"Siempre que sea posible, las funciones de seguridad suplementarias de una capa deben definirse de modo que no se impida la realización* (de dichas funciones suplementarias) *en forma de uno o varios módulos autónomos."* El grado de independencia perseguido por esta recomendación se puede conseguir con un desacople efectivo de la seguridad del nivel de aplicación como el que propone IoTsafe. Todas las funcionalidades de seguridad que ofrece actualmente se consiguen combinando elementos software independientes entre sí, como el protocolo de seguridad empleado o el framework de red. Esto permite que:

- Existan diferentes protocolos de seguridad operando en la misma infraestructura IoT (diferentes versiones de SSH u otro protocolo de proxificación).
- Se puedan utilizar diferentes versiones de *kernel* o de Netfilter.
- Se pueda complementar la seguridad de IoTsafe con módulos de seguridad LSM como AppArmor o SeLinux.
- Resulte más sencilla la utilización de sistemas de IA de seguridad complementarios basados en técnicas de DL o DPI, que contribuyan a mejorar la seguridad de IoTsafe gracias al tráfico en claro de paquetes entre interfaces de red protegidas.

h) "*Debe poder aplicarse la presente Recomendación a los sistemas abiertos constituidos por sistemas extremos que contienen las siete capas y a los sistemas relevadores."* El sistema propuesto por IoTsafe persigue precisamente que otros sistemas OSI puedan incorporar este esquema de comunicación sin necesidad de modificar la implementación de sus sistemas respectivos, al apoyarse en tecnología software madura de uso general y basar el desacoplo de la seguridad de las aplicaciones en la API de *socket*.

3.4.1. **Encaje de IoTsafe en los niveles OSI**

Los dispositivos gestionados por IoTsafe ejecutan las aplicaciones IoT de tal forma que los procesos relativos a IoTsafe resultan totalmente transparentes. Si se analizan estos dispositivos IoT externamente, la percepción que se tiene de IoTsafe queda representada en la Fig. 3.4-1.

La estructura y funcionamiento de IoTsafe inducen a pensar que los servicios que éste ofrece a las aplicaciones se sitúan inmediatamente por debajo de ellas, pero por encima del protocolo de nivel de Transporte (TCP/UDP). IoTsafe resulta totalmente transparente a la aplicación IoT, pero esto resultaría contradictorio con un principio básico del modelo de referencia OSI: por definición, una capa OSI no puede ser transparente a todas las demás capas, sino que debe ser utilizada explícitamente por la inmediatamente superior [117].

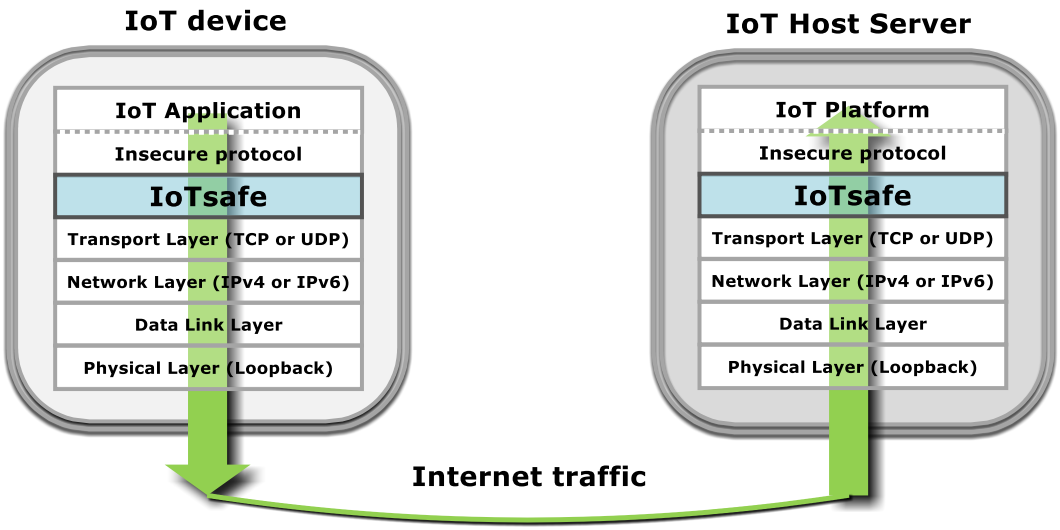


Fig. 3.4-1: Encaje de IoTsafe según el modelo OSI si se analiza el dispositivo IoT externamente. Al presentar un desarrollo basado en la pila TCP/IP, la capa de sesión y presentación tradicionalmente absorbidas por la aplicación podrían delegar las responsabilidades de seguridad y autenticación en IoTsafe. El término "Inecure protocol" hace referencia a una versión insegura de protocolo de aplicación: HTTP, CoAP, MQTT, etc.

Sin embargo, si se analiza el dispositivo IoT desde su interior, IoTsafe no funciona como una capa OSI estándar, sino como un "*sistema abierto relevador*"[92] de una capa por encima de la de transporte [117] (Fig. 3.4-2).

Utilizar este tipo de "*sistema relevador*" a nivel interno tiene cierto sentido si se considera que los requisitos de seguridad son diferentes en el exterior y en el interior del dispositivo. De ese modo, mientras que la aplicación IoT forma, junto con el resto de capas, un "*sistema de extremo OSI*"[93], IoTsafe funciona en los dispositivos como un "*sistema abierto relevador*" o un "*sistema de retransmisión (N) OSI*"[94], lo cual justifica la transparencia que caracteriza a IoTsafe a nivel de aplicación, al operar como *sistema abierto relevador de nivel 4* (Fig. 3.4-3).

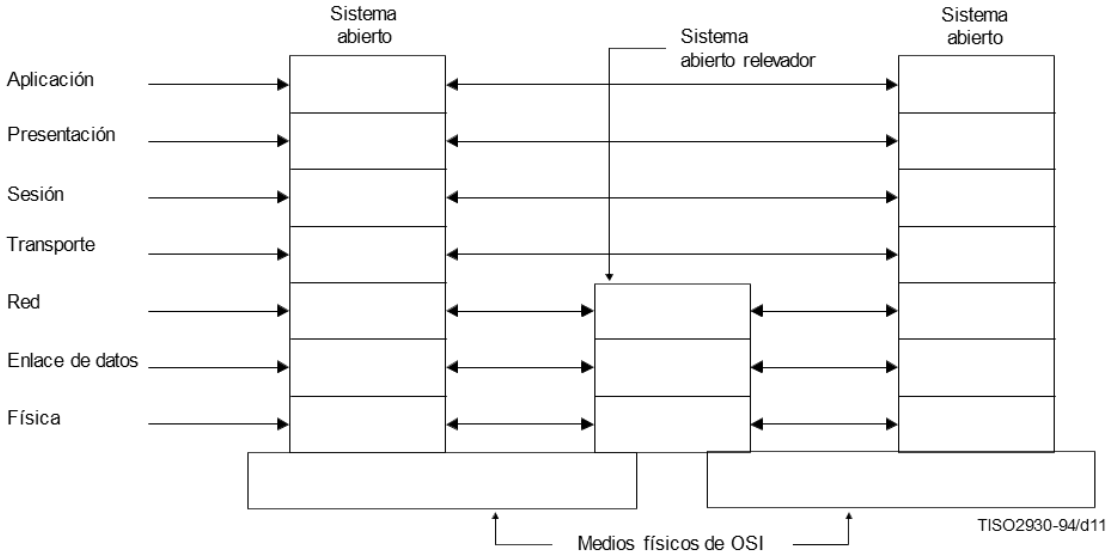


Fig. 3.4-2: Comunicación entre dos sistemas abiertos por medio de un "sistema abierto relevador de nivel de red" [117].

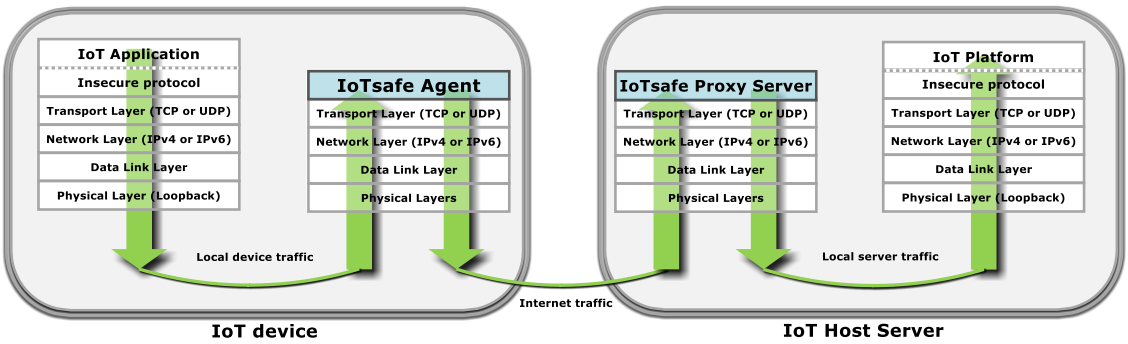


Fig. 3.4-3: Encaje de IoTsafe según el modelo OSI como un sistema relevador de nivel 4 si se analiza su estructura desde dentro de los dispositivos IoT.

[92] **Sistema abierto relevador**: Un tipo de sistema abierto[91] que sirve de intermediario entre dos sistemas abiertos que no comparten un medio físico OSI común [113].

[93] **Sistema de extremo OSI**: *"Sistema abierto que, para un caso de comunicación determinado, es el último origen o destino de datos."* [113].

[94] **Sistema de retransmisión (N) OSI:** *"Sistema abierto que, para un caso de comunicación determinado, utiliza funciones OSI hasta funciones de la capa (N) inclusive y cuando se realiza una función de retransmisión en la capa (N)."* [113].

Una configuración similar a la que presenta IoTsafe puede encontrarse en despliegues de red que empleen tecnologías VPN. Las arquitecuras VPN no tienen que conformar necesariamente una capa OSI, sino que en algunas circunstancias pueden entenderse también como un sistema OSI relevador, integrado en los dispositivos de forma similar al seguido por IoTsafe, pero trabajando en niveles OSI inferiores (nivel de red, o nivel 3). OpenVPN es un claro ejemplo de este último caso, tal y como puede observarse en la Fig. 3.4-4.

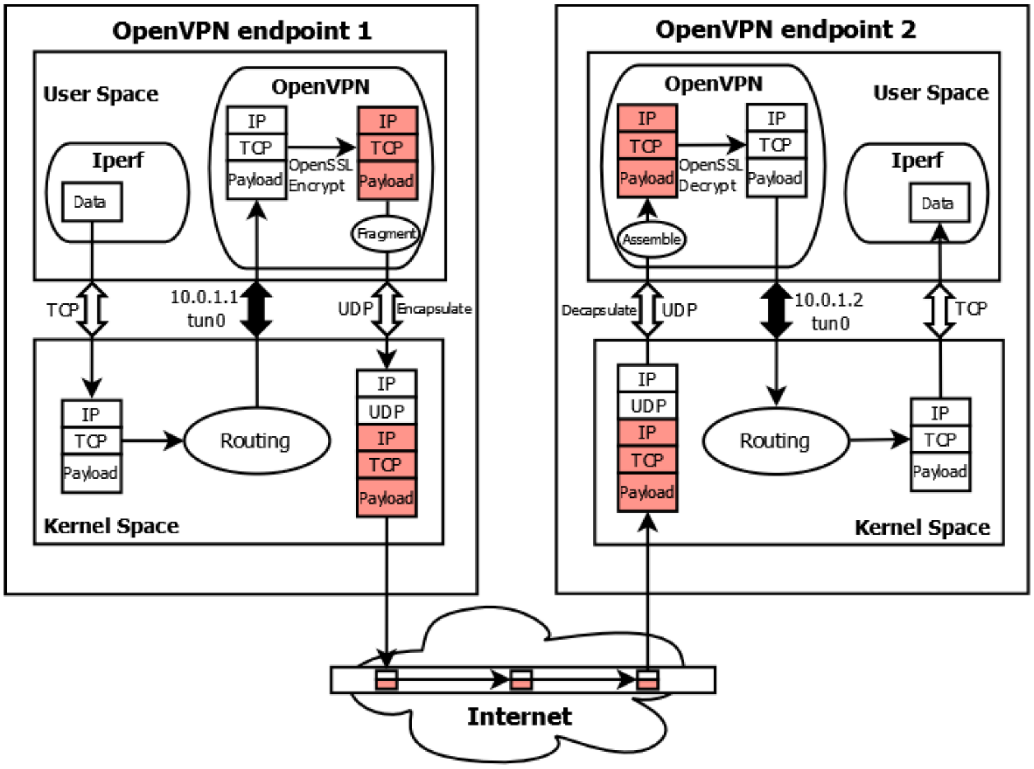


Fig. 3.4-4: Flujo de datagramas en OpenVPN [119]***. En el ejemplo se aprecia el uso del encapsulado característico de la tunelado y de la existencia de dos tipos de sistemas OSI diferenciados en el que uno de ellos actúa como relevador a nivel 3 OSI.***

Las principales diferencias entre OpenVPN e IoTsafe radican en el nivel en que se produce este enrutamiento: En OpenVPN es en el nivel 3, mientras que en IoTsafe es en el nivel 4 (a nivel de *socket*). Esto explica el hecho de que en OpenVPN se realice tunelado del tráfico, encapsulando datagramas completos, mientras que en IoTsafe no existe encapsulación. Esto se refleja en el hecho de que IoTsafe envíe y reciba únicamente el *payload* de nivel de transporte, existiendo para ello tres segmentos de comunicación bien diferenciados (Fig. 3.2-8), fruto de comunicaciones basadas en proxificación (Fig. 2.3-9 y Fig. 2.3-10).

Por tanto, IoTsafe puede entenderse como una abstracción del espacio topológico de comunicaciones que mantiene las propiedades OSI de las capas, a la par que disminuye los requisitos de complejidad (red, seguridad y autenticación). Esta abstracción permite unificar procesos involucrados en las comunicaciones de diferentes arquitecturas, topologías y

despliegues tecnológicos, sin violar el sistema OSI que emplee la aplicación IoT. De este modo, el dispositivo IoT, independientemente de la arquitectura, el despliegue de red y el diseño, accederá a los recursos de la plataforma IoT de forma local, como si estuviesen dentro del propio dispositivo. Los esquemas basados en VPN ofrecen servicios parecidos, interconectando a distancia redes locales distintas y de forma transparente. La diferencia en este caso radica en que en IoTsafe, los recursos ofrecidos son *socket* de comunicación locales (proxificaciones) adecuadamente protegidos en origen y destino, de manera que la comunicación entre la aplicación IoT y la plataforma IoT conserva la seguridad en todo momento gracias al establecimiento de asociaciones de seguridad entre estos dos sistemas, en forma del contexto de seguridad definido por IoTsafe. Los sistemas VPN sólo gestionan la seguridad relativa a la interconexión de las redes.

3.4.2. El contexto de seguridad definido por IoTsafe estudiado según las recomendaciones de la UIT

Según la recomendación X.803 de la UIT [120] en relación a los modelos de seguridad en capas superiores, una *"asociación de seguridad"*[95] existente entre dos entidades, supone la existencia de reglas de interacción seguras y el mantenimiento de un estado de seguridad coherente en ambos sistemas. Una *"asociación de seguridad con transmisión"*[96] se considera un caso particular de este concepto, en el que puede haber, entre otros eventos, una retransmisión de las comunicaciones de la aplicación. Este escenario es el que puede representar mejor el funcionamiento básico de las comunicaciones proxificadas en IoTsafe. En el esquema de comunicación propuesto, se desacopla la seguridad de la aplicación, dando lugar a dos sistemas diferenciados: el de la aplicación IoT junto con la plataforma IoT; y el de IoTsafe como sistema relevador. El establecimiento de una asociación de seguridad entre ambos se produce por medio del "contexto de seguridad" descrito en la sección 3.2.2.

Ese concepto de contexto de seguridad, puede servir para desarrollar en IoTsafe la recomendación UIT-T X.803 dedicada al análisis de las funciones de seguridad asociadas a las capas superiores [120]. En la Fig. 3.4-5 se describe el esquema de prestación de servicios OSI característico para capas superiores. En este esquema, los "*objetos de seguridad de sistema*" (SSO) representan *"un conjunto de funciones conexas de seguridad de sistema"*. Las "*funciones de seguridad de sistema*", a su vez, consisten en "*todas aquellas actividades relacionadas con la seguridad, tal como el cifrado/descifrado, firma digital o la generación o procesamiento de un certificado o testigo de seguridad transmitido en un intercambio de autenticación*".

[95] **Asociación de seguridad:** *"Es una relación entre dos o más entidades para la que existen atributos (información y reglas de estado) que rigen la prestación de servicios de seguridad que afectan a esas entidades."* [116].

[96] **Asociación de seguridad de transmisión:** *"Asociación de seguridad entre dos sistemas para sustentar la comunicación protegida a través de una retransmisión de aplicación (por ejemplo, en aplicaciones con almacenamiento y retransmisión, o con encadenamiento)."* [116].

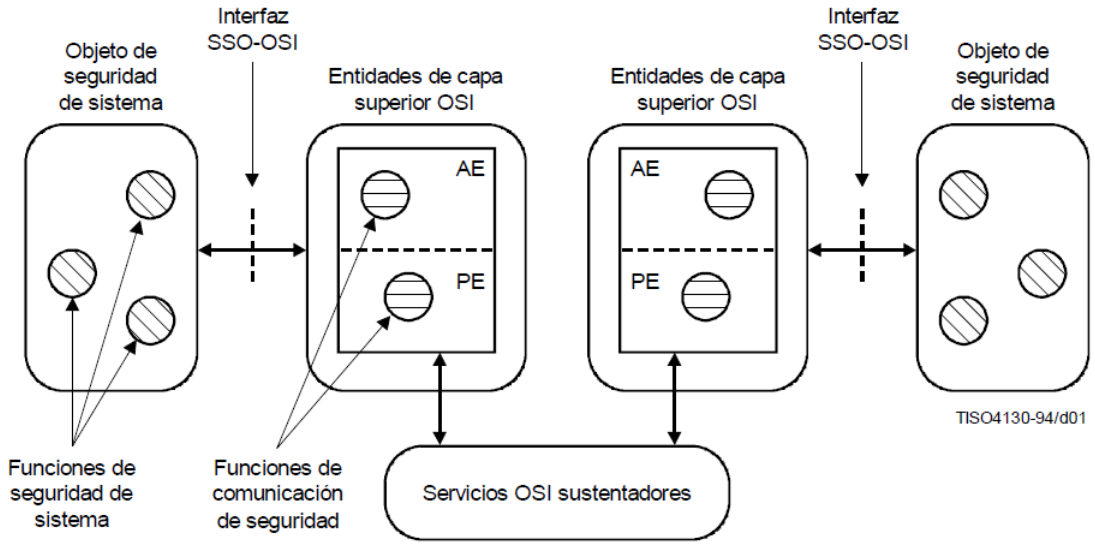


Fig. 3.4-5: Funciones de seguridad asociadas con las capas superiores de OSI [120].

Las entidades de capa superior (presentación y aplicación) pueden hacer uso de estas "*funciones de seguridad de sistema*" a través de una interfaz SSO-OSI. Para ello, existen "*funciones de comunicación de seguridad*" que permiten definir un protocolo que establezca la seguridad a nivel de aplicación empleando dichas "*funciones de seguridad del sistema*".

En IoTsafe, se podría extender el concepto de "*funciones de seguridad del sistema*" para incluir todas aquellas definidas por IoTsafe para establecer los contextos de seguridad, los *socket* protegidos, solicitar las proxificaciones, etc. (Fig. C-18). Esta medida tiene sentido una vez se asume el desacoplo de seguridad en el modelo de referencia, ya que surgen nuevas funciones de seguridad del sistema, necesarias para gestionar los contextos de seguridad (que son manejados por el sistema operativo).

Las "*Funciones de comunicación de seguridad*" son aquellas invocadas por la capa de aplicación o presentación para configurar la ejecución de las "*funciones de seguridad del sistema*", en base a las necesidades de comunicación y seguridad de la aplicación. El objeto de los dispositivos IoT suele estar bien determinado por cometidos concretos y necesidades de comunicación y seguridad específicas, que normalmente son invariantes (contador de agua, medidor de presión arterial, lector de tarjeta de identificación, sensores inteligentes, etc). Esto permite eximir a la capa de aplicación de la lógica requerida para incorporar las "*funciones de comunicación de seguridad*", volviéndose transparente a IoTsafe en los dispositivos IoT. Esto se podría automatizar durange el arranque del dispositivo, cuando se produce el establecimiento de las comunicaciones pertinentes por parte del agente_B de IoTsafe.

Desde el punto de vista de la recomendación X.803 y teniendo en cuenta el cometido invariante de los dispositivos IoT, se podría entender al sistema operativo de los dispositivos IoT operando bajo IoTsafe, como una sencilla capa de presentación de la seguridad de las comunicaciones para la capa de aplicación, de manera que las funciones de comunicación de

seguridad se invocan invariantemente de la misma manera en todas las ocasiones. Por tanto, la capa de aplicación no dispondría de "*funciones de comunicación de seguridad*" al resultar IoTsafe transparente a esta última.

La interfaz SSO-OSI en los dispositivos IoT es muy simple y se corresponde con el *framework* de red y el control que éste ejerce en los flujos de comunicaciones. La inicialización del *framework* de red se puede planificar en el arranque y no requiere de cambios de configuración posteriores, por la invariabilidad en el objeto y en el cometido de los dispositivos.

En el servidor IoT, IoTsafe ya no resulta transparente a la plataforma. Ésta debe ser consciente de la existencia de IoTsafe para poder hacer uso de las "funciones de comunicación de seguridad" necesarias, que invoquen las "funciones de seguridad del sistema" pertinentes, para habilitar según requiera la configuración de la seguridad de las comunicaciones. La interfaz SSO-OSI en este caso resulta muy dinámica. Netfilter activa y desactiva el acceso a los *socket* protegidos, según se necesite por la capa de presentación y aplicación: servidor *proxy-firewall* y la plataforma IoT respectivamente.

3.4.3. **Ventajas de seguridad complementarias derivadas de utilizar IoTsafe según la UIT: Seguridad pervasiva**

La estrategia de diseñar IoTsafe como un sistema relevador permite profundizar en la seguridad pervasiva sin incrementar la complejidad de las aplicaciones IoT de los dispositivos. Los mecanismos de seguridad pervasivos no forman parte de una capa determinada, y pueden considerarse como aspectos de gestión de seguridad complementarios [118]:

- **Funcionalidad de confianza**: "*La funcionalidad de confianza puede utilizarse para ampliar el campo de aplicación o para establecer la eficacia de otros mecanismos de seguridad. Toda funcionalidad que proporciona directamente los mecanismos de seguridad o que permiten el acceso a estos mecanismos deberá ser digna de confianza.*"… "*En general, estos procedimientos son costosos y difíciles de aplicar. Los problemas pueden minimizarse eligiendo una arquitectura que permita realizar funciones de seguridad en módulos que puedan estar separados de las funciones no relacionadas con la seguridad o proporcionadas por éstas*." A este respecto, se puede afirmar que el esquema propuesto por IoTsafe contribuye a reforzar la funcionalidad de confianza, al fomentar el desacoplo de la seguridad de las aplicaciones.
- **Etiquetas de seguridad**: "*Los recursos que comprenden elementos de datos pueden tener asociadas etiquetas de seguridad, por ejemplo, para indicar un nivel de sensibilidad. A menudo es necesario transportar la etiqueta de seguridad apropiada con datos en tránsito. Las etiquetas de seguridad explícitas deben ser claramente identificables, para poder verificarlas de manera apropiada. Además, deben estar vinculadas de una manera segura a los datos con los cuales están asociadas.*" IoTsafe

desarrolla en los contextos de seguridad, un esquema de etiquetas de seguridad en forma de marcado de paquetes que permite, junto con las reglas de filtrado de Netfilter, operar adecuadamente las reglas de seguridad relativas a los flujos de datos.

- ***Detección de eventos****: "La detección de eventos relativos a la seguridad comprende la detección de violaciones aparentes de seguridad y puede incluir también la detección de eventos «normales» tales como el acceso logrado (o «log on»). Los eventos relativos a la seguridad pueden ser detectados por entidades, dentro de la ISA* (Interconexión de Sistemas Abiertos)*, que comprenden mecanismos de seguridad.*" Netfilter, a través de los *hooks* del *kernel* [121] y de los framework LSM que los gestiona (por ejemplo, SELinux o AppArmor), es capaz de detectar fácilmente estos eventos. IoTsafe, al incorporar Netfilter en su esquema, permite el desarrollo de módulos independientes de detección de eventos que enriquezcan el grado de seguridad pervasiva ofrecida por IoTsafe en este aspecto (Fig. C-17).
- ***Registro de auditoría de seguridad****: "Los registros de auditoría de seguridad proporcionan un mecanismo de seguridad valioso dado que hacen posible detectar e investigar potencialmente las violaciones de seguridad permitiendo una auditoría de seguridad posterior"*[97]. Por su diseño, IoTsafe permite servicios complementarios que se apoyen en la detección de eventos y el desacoplo de la seguridad. De este modo, se pueden elaborar procedimientos estandarizados de auditoría que validen el nivel de seguridad ofrecido por un dispositivo, sin necesidad de tener que analizar aspectos de diseño de la aplicación IoT del propio dispositivo: la seguridad está desacoplada de las aplicaciones y se puede proceder con "certificaciones compuestas", que resultan más económicas, rápidas y fáciles de estandarizar. Este tipo de certificaciones más flexibles, rápidas, económicas y amparadas por la ISO 15408-1[98], basan la certificación del producto final en la certificación de las partes del mismo, al considerarlo un "*composed assurance package*"[99], junto con un procedimiento de fabricación concreto y específico que también es auditado. Esta idea de certificación por bloques, que aumenta las posibilidades de reutilización, ya se viene recomendando por ENISA específicamente para ecosistemas IoT (en el anexo B.4 de [122]).
- ***Recuperación de seguridad****: "La recuperación de seguridad trata las peticiones provenientes de mecanismos tales como las funciones de tratamiento y de gestión de los eventos y realiza acciones de recuperación como resultado de la aplicación de un conjunto de reglas. Estas acciones de recuperación pueden ser de tres tipos:*

[97] **Auditoría de seguridad***: "Estudio independiente y un examen de las anotaciones y de las actividades del sistema para probar la idoneidad de los controles, asegurar la coherencia con la política establecida y con los procedimientos de explotación, ayudar a evaluar los daños y recomendar modificaciones de los controles, de la política y de los procedimientos."* [114].

[98] "ISO/IEC 15408-1:2009 Information technology — Security techniques — Evaluation criteria for IT security — Part 1: Introduction and general model", *ISO Online Browsing Platform (OBP)*, 2009, https://www.iso.org/obp/ui/#iso:std:iso-iec:15408:-1:ed-3:v2:en

[99] "Common Criteria for Information Technology Security Evaluation. Part 3: Security assurance components" *Common Criteria Portal*, 2009, https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R3.pdf

a) *Inmediatas (ej.: provocar un aborto inmediato de las operaciones, como una desconexión);*
b) *Temporales (ej.: producir la invalidación temporal de una entidad); y*
c) *A largo plazo (ej.: la inclusión de una entidad en una «lista negra» o el cambio de una clave).”*

El control de las comunicaciones ofrecido por IoTsafe permite desarrollar mecanismos de este tipo que refuercen la seguridad pervasiva, gracias a la utilización de las comunicaciones proxificadas por protocolos seguros como SSH. Además, al poder monitorizar el tráfico en el servidor sin ningún tipo de seguridad (Fig. 3.4-6), vuelve más sencillos de detectar los comportamientos anómalos, gracias a técnicas basadas en DPI y DL. Esta estrategia es particularmente útil en el caso en que un dispositivo concreto sea comprometido o infectado, y utilice sus comunicaciones proxificadas para lanzar ataques desde dentro del servidor IoT. Pese a que el diseño de los contextos de seguridad limita la mayoría de los posibles ataques (1, 3, 4 y 5 en la Fig. 3.1-18), algunos de denegación de servicio (por ejemplo, el número 2 en Fig. 3.1-18) necesitan de este tipo de tecnología de seguridad complementaria que permita detectar comportamientos anómalos de dispositivos aparentemente confiables. De este modo, se podrán desarrollar algoritmos de detección temprana de este tipo de ataques, que permitan mitigar sus efectos con rapidez.

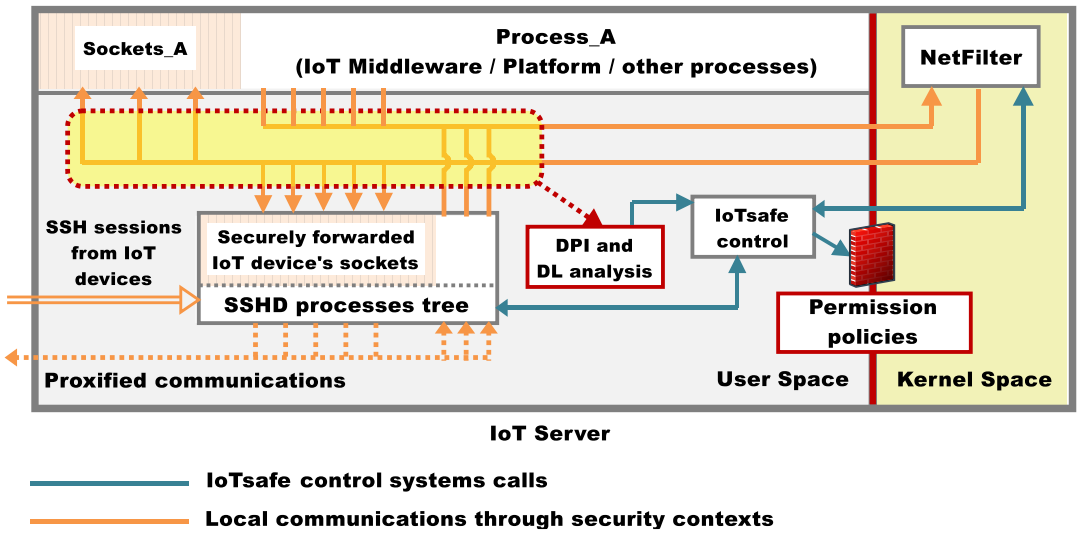


Fig. 3.4-6: Diagrama conceptual simplificado de un gateway/server de IoTsafe en el que el tráfico local es analizado usando técnicas DPI y DL para acelerar la detección posibles ataques.

3.4.4. IoTsafe: hacia un framework del nivel de Transporte

Se puede decir que el nivel de Transporte se ha ido “fosilizando” con el tiempo, dificultando el que se produzcan evoluciones de calado en este ámbito [123]. Esto es en parte consecuencia

del despliegue de *middleboxes*[100] que lastran el desarrollo de nuevas técnicas de transporte. Por otro lado, la tímida evolución de la API de transporte limita la evolución de las funcionalidades de comunicación que se pueden presentar a las aplicaciones. En [123] se realiza un estudio detallado de las problemáticas derivadas de esta realidad y de posibles estrategias para abordarla. Entre ellas, una de la más realistas consiste en dar por hecho que el estatus del nivel de transporte seguirá así por mucho tiempo y, por ello, tratar de desarrollar de forma superpuesta un framework de transporte que permita paliar las necesidades y problemas encontrados. Esta estrategia persigue varios objetivos [123], muchos de ellos ya abordados por IoTsafe:

Flexibilidad de la API: "*La API* (de socket) *debe ser capaz de desacoplar las aplicaciones de funcionalidades y protocolos inferiores*."

- Retro compatibilidad: "*Se debe facilitar que aplicaciones que no hicieron uso de la API en su desarrollo, hagan uso del framework facilitando su adopción progresiva*." IoTsafe permite este aspecto al adoptar íntegramente la API de *socket* estándar y no aplicar modificaciones de ningún tipo a la misma, permitiendo que cualquier infraestructura IoT que emplee la API de socket pueda incorporar muy fácilmente IoTsafe.
- Soporte de configuración de aspectos de bajo nivel: "*La API debe permitir configurar aspectos de bajo nivel de las comunicaciones* (parámetros de los protocolos utilizados)". IoTsafe no ofrece soporte en este aspecto, más allá del que ofrece la API actual. Existen características muy generales de la API de socket, que son aplicables a todas las comunicaciones (algunos parámetros TCP y el algoritmo de control de congestión). Para desarrollar esta característica y permitir profundizar en la personalización del funcionamiento de la API en cada conexión, probablemente sería necesario combinar funcionalidades de MultiStack[101] en IoTsafe (Fig. 3.4-7).

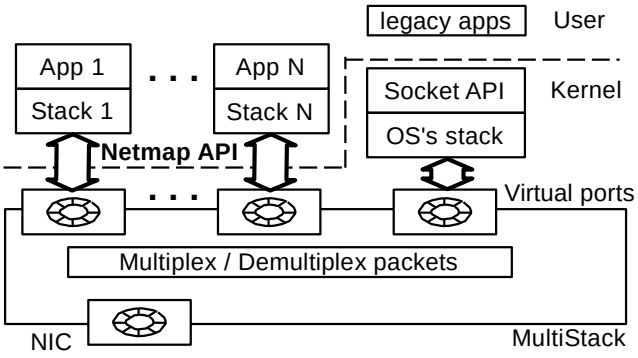


Fig. 3.4-7: Arquitectura software de MultiStack [124]***. IoTsafe podría combinar este esquema en el servidor proxy-firewall para adaptar el stack a las proxificaciones de cada dispositivo adecuadamente y poder ofrecer servicios de transporte complementarios.***

[100] **Middle-boxes**: Dispositivos que encaminan paquetes IP entre redes, bloquean trafico malicioso, realizan traducción de direcciones de red (NAT), mejoran el rendimiento y en algunos casos realizan tareas de deep inspection entre otros [181]

[101] M. Honda, "*MultiStack – Kernel Support for Multiplexing and Isolating User-space Stacks*", *Github*, 2015, https://github.com/sysml/multistack

- Soporte de configuración de aspectos de alto nivel: "*También se debe proporcionar la capacidad de configurar aspectos de la comunicación como el control de latencia, calidad de servicio, soporte de movilidad, etc*." IoTsafe no proporciona directamente esta funcionalidad, pero puede desarrollarse para dar soporte al respecto a través de los perfiles de conexión. En el Anexo C ejemplo 3 se presenta el desarrollo de una funcionalidad complementaria a través de una autenticación transparente, indicada en el perfil de conexión empleando los perfiles de conexión definidos por IoTsafe.
- Comprensibilidad: "*Se debe facilitar que los aspectos relativos a la comunicación que el framework haya podido definir automáticamente sean accesibles a instancias superiores si se requiere conocerlos"*. IoTsafe ya implementa esta característica parcialmente en el servidor, a través del *connection-broker* (Fig. C-11).

Facilidad de despliegue: "*El framework debe poderse incorporar con facilidad en los sistemas minimizando el impacto de las actuaciones necesarias*."

- Caracterización como aplicación: "*El framework debe tratar de resultar lo más independiente posible del sistema operativo subyacente*." Esto es algo intrínseco al diseño de IoTsafe para conseguir un desacoplo de seguridad efectivo. Se consigue definiendo módulos de seguridad *stand-alone,* con mínimas dependencias con el *kernel* de Linux y mantienendo la portabilidad binaria entre plataformas de misma arquitectura.
- Tolerancia a la diversidad de características de los OS: "*El framework debe estar diseñado de manera que permita la integración de distintos protocolos y funcionalidades disponibles en el sistema operativo para aprovecharlas en caso de estar disponibles*." La arquitectura de IoTsafe permite desarrollar en un futuro la lógica necesaria para incorporar esta característica, aunque actualmente no lo permite (ej.: *Stream Control Transmission Protocol,* SCTP [125] o *Datagram Congestion Control Protocol*, DCCP [126]).
- Tolerancia a la diversidad de los dispositivos origen y destino de la comunicación: "*No se puede asumir que los extremos siempre usen el nuevo framework o que este sea de la misma versión. Aun bajo el riesgo de poder ofrecer menores prestaciones, resulta fundamental poder ofrecer este tipo de tolerancia para permitir un desarrollo gradual y una implementación incremental.*" IoTsafe no consiste en una implementación concreta de una tecnología, sino que describe un método y sistema de comunicación que constituye la base para el desarrollo del nuevo *framework*. Esto permite que, por ejemplo, pueda utilizarse SSH en un dispositivo_B, y otro protocolo de proxificación en otro dispositivo_B distinto, que ambos se conecten al mismo servidor (dispositivo_A) y disfruten de los servicios de comunicación de forma transparente entre ellos. IoTsafe únicamente opera con proxificaciones en el servidor (socket), y las plataformas IoT que necesiten conectarse a los dispositivos IoT son totalmente ajenas a la implementación del protocolo de proxificación.

- Tolerancia a la diversidad de las redes: "*La posibilidad de utilizar el framework no debe depender del tipo de red empleada o de los servicios que esta ofrece y en particular del tipo y funcionalidades ofrecidas por los middleboxes intermedios.*" IoTsafe no basa su funcionamiento en ninguna característica especial de la red o de los middleboxes, por lo que el futuro desarrollo del *framework* de transporte podrá elaborar optimizaciones a este respecto como servicios estrictamente opcionales. Las comunicaciones M2M se realizan siguiendo un esquema de comunicación que utiliza un servidor intermedio (el servidor IoT), evitando técnicas de comunicación NAT trasversal basadas en *Session Traversal Utilities for* NAT (STUN) [127], *Traversal Using Relays around* NAT (TURN) [128], *Interactive Connectivity Establishment* (ICE) [129] [130] o en *hole punching* [131].

Extensibilidad: *El framework presentado debe soportar de forma transparente la evolución independiente de los modulos que lo compongan.*

- Soporte en la evolución del *framework*: "*Un framework evolutivo debe permitir la inclusión de nuevos tipos de protocolos y de características técnicas en el futuro*." IoTsafe se presenta como una tecnología totalmente transparente. Esta prestación se basa en el desacoplo, no sólo de la seguridad, sino también de las funciones de comunicación entre dispositivos. Esta estrategia facilita un soporte futuro ante la evolución.
- Soporte en la evolución del sistema operativo: "*La interfaz entre el framework y el sistema operativo puede cambiar con el tiempo para mejorar el servicio provisto por el propio framework. Esto permite que los dispositivos y los servidores que implementen el framework puedan evolucionar también*." IoTsafe se presenta con un muy alto grado de independencia sobre el sistema operativo y este hecho, a su vez, le permite tolerar fácilmente cualquier evolución experimentada a lo largo del tiempo.

Selección paramétrica guiada: *La pila actual de protocolos de red y transporte require elegir parámetros explícitos para la selección de los valores. Por ejemplo, una aplicación puede elegir entre IPv4 o IPv6 y seleccionar DCCP, SCTP, TCP, UDP-Lite o UDP. Además, los parámetros pueden ser especificados a nivel de socket o de protocolo. El Framework debería ser capaz de combinar la información de toda la red y local para permitir una selección óptima de los valores adecuados de la comunicación que permitan satisfacer los requisitos de la misma.* IoTsafe no incluye ninguna funcionalidad de este tipo, más allá de la selección de parámetros aceptados por el perfil de conexión. Sin embargo, es una característica que podría desarrollarse de forma complementaria en un futuro sin afectar a su funcionamiento estándar.

- Mapeo de valores paramétricos: *"El framework debe poder mapear requerimientos de alto nivel proporcionados por la aplicación en requisitos de más bajo nivel. Este tipo de selección de parámetros debería ser guiada por los requerimientos característicos de cada aplicación, teinendo como resultado la selección de interfaces a utilizar, el tipo de*

protocolo de red, y de transporte, y la asignación de parámetros de red en cada capa". IoTsafe implementa los perfiles de conexión (Perfil_A, Perfil_B, etc) que se configuran para cada aplicación y comunican al agente las necesidades de la aplicación (*sockets* a proxificar, dominio del servidor IoT, etc). Este elemento se puede extender para dar cobertura a otros requerimientos de alto nivel que presente la aplicación y que tengan que ser puestos en marcha por IoTsafe.

- Detección de funcionalidades del SO: *"Si es possible, las herramientas del sistema operativo que proporcionan información de interés relativa a la comunicación, deben poder ser aprovechadas por el framework a la hora de tomar decisiones para la selección de los parámetros y valores."* IoTsafe no se ha desarrollado con este tipo de prestaciones, pero se pueden entender como funcionalidades complementarias potenciales.
- Detección de funcionalidades de red y de los dispositivos extremos de la comunicación: *''Cualquier decision que utilize un protocol particular debe basarse en el set de protocolos utilizados por ambos extremos de la comunicación. Como prerrequisito a utilizar uno de esos protocolos, se debe incluir también el soporte de los middleboxes que existan en el camino."* El *framework* debería ser capaz de soportar mecanismos de descubrimiento de las características end-to-end y de las prestaciones remotas disponibles. Actualmente, IoTsafe no soporta este tipo de prestaciones de optimización, pero al considerar los flujos de las comunicaciones como circuitos virtuales independientes, se puede facilitar el desarrollo de un tratamiento diferenciado que se beneficie de prestaciones existentes en la red.
- Capacidad de adaptación de los parámetros de comunicación en el tiempo: *"La decisión final de utilización de los parámetros de comunicación, debe basarse también en la información histórica relativa a las comunicaciones utilizadas en ese camino, pero también debe considerar que dichas características pueden cambiar en el tiempo (a largo y a corto plazo)".* Respecto a IoTsafe, este tipo de funcionalidades se pueden diseñar en una fase más avanzada del desarrollo del *framework* de transporte. IoTsafe emplea perfiles de conexión que pueden evolucionar para dar soporte a nuevos servicios como este.
- *Transparente a los protocolos de aplicación: "El testeo y descubrimiento llevado a cabo por el framework de transporte debe realizarse por dicho framework y no debe requerir de cambios o soporte específico por los protocolos de aplicación."* IoTsafe, pese a no incorporar funcionalidades de descubrimiento de prestaciones de red, sí defiende la independencia respecto de la aplicación e incluye otras funcionalidades que se sustentan en este principio, como las del ejemplo presentado en el Anexo E3. En dicho ejemplo se presenta una implementación de IoTsafe complementada con un módulo extra (Proxy HTTPS⇔HTTP), que permite unas comunicaciones transparentes entre dispositivos de gestión (usando HTTPS) y dispositivos IoT (usando HTTP) de forma

totalmente transparente para ambos. Ese ejemplo demuestra la capacidad de IoTsafe para desarrollar servicios complementarios que cumplan con las exigencias de transparencia previstas en este punto.

Escalabilidad: *"El framework debe sere ser escalable en diversos aspectos:"*

- Volumen de tráfico: *"El framework debe limitar el impacto en el consumo de CPU y poder distribuir la carga entre los recursos existentes para soportar altos volúmenes de tráfico. Siempre que sea posible debe tratarse de utilizar cualquier soporte hardware disponible para estas tareas. Al mismo tiempo, el propio framework debe tratar de evitar tráfico propio de señalización que limite la escalabilidad."* IoTsafe contempla estrategias de escalado en relación al volumen de tráfico (Capítulo 4). También es capaz de emplear *hardware* dedicado de cifrado y de compresión si el servidor *proxy-firewall stand-alone* y su agente (servidor SSH y cliente) se compilan con las librerías que den soporte especifico para ello (*hardware encription engines*[102], *hardware* ZLIB *library*[103]).
- Número de comunicaciones y dispositivos: *"El número de asociaciones de transporte requeridas (conexiones TCP, asociaciones SCTP o flujos UDP) depende de cada tipo de caso. El framework debe soportar eficientemente un número elevado de comunicaciones de transporte simultaneas."* IoTsafe cuenta con estrategias de multiplexión de canales en cada sesión, que permiten balancear la carga entre diferentes sesiones para optimizar el uso de los recursos disponibles (Sección 4.4).
- Número de prestaciones: *"Finalmente, el framework, una vez desarrollado, debe ser capaz de dar soporte a una amplia variedad de combinaciones de parámetros, configuraciones e interacciones de red. El proceso de selección de los parámetros óptimos debe resultar aceptable para un establecimiento de comunicación."* IoTsafe delega la mayor parte de los aspectos del establecimiento de las comunicaciones al protocolo de proxificación subyacente que se elija (normalmente SSH). IoTsafe toma, junto a este protocolo, las consideraciones necesarias para establecer las proxificaciones que se emplean durante las comunicaciones. Este proceso se podrá modular automáticamente en un futuro por los servicios opcionales del *framework*. Por otro lado, este proceso no supondrá una sobrecarga elevada en el sistema, ya que se prevé que se realice con baja frecuencia: las proxificaciones establecen contextos de seguridad sobre sesiones del protocolo de proxificación que deben permanecer en el tiempo. Así se evita tener que establecer conexiones o intercambiar parámetros de las comunicaciones con demasiada periodicidad, evitando que se afecte negativamente a los tiempos de respuesta, *overhead* y consumo energético.

[102] "OpenSSH v.4.6 (con soporte para motores criptográficos hardware)", 2006, https://www.openssh.com/txt/release-4.4
[103] IBM, "Integrated Accelerator for zEnterprise Data Compression", 2019 https://www.ibm.com/downloads/cas/AE7VLK0V

IoTsafe ha sido planteado para ofrecer un desacoplo en la seguridad de las aplicaciones. Para diseñar este método de comunicación, se ha tenido que cumplir con muchos de los aspectos arriba mencionados. El cumplimiento de aquellos aspectos que resultan más importantes facilita la evolución de IoTsafe de manera que éste pueda sumar esfuerzos junto con otras iniciativas y proyectos diferentes que trabajen por la consecución de algún tipo de *framework* similar al que se describe en este apartado.

3.5. PRINCIPALES VENTAJAS DE IOTSAFE

En primer lugar, consideramos que la mayor ventaja del método y sistema de comunicación, presentado en la sección 3.2, es que desacopla la seguridad respecto de las aplicaciones y permite que las certificaciones inherentes a dicha seguridad y sus comunicaciones puedan tratarse de forma independiente del resto del *software* del dispositivo y del servidor. Esto permite facilitar el cumplimiento con regulaciones derivadas del GDPR [14], la directiva de privacidad electrónica [132], la regulación de privacidad electrónica [133], las directivas NIS [134], las iniciativas llevadas a cabo por la *Cybersecurity and Infrastructure Security Agency* (CISA)[104] y otras regulaciones emergentes como la de California SB-327 [135] para ciberseguridad y dispositivos IoT.

Por sí mismo, IoTsafe no ofrece ningún procedimiento de certificación ni garantiza por sí solo el cumplimiento de las regulaciones arriba presentadas. Sin embargo, su estructura sí permite abordar esas problemáticas de una manera más generalista y estandarizada, proporcionando entre otras las siguientes ventajas [108] [107]:

- **Reducción de costes en el desarrollo del *software* IoT**: Los aspectos de seguridad se delegan a IoTsafe.
- **Aumento de la vida útil de los dispositivos IoT y de su *software***: Si el dispositivo queda obsoleto por motivos de seguridad, es más factible su actualización, al estar dicha seguridad desacoplada de las aplicaciones principales del dispositivo IoT.
- **Facilidad en el proceso de horizontalización de *verticals***: Resulta relativamente sencillo intercomunicar dispositivos con plataformas IoT diferentes. IoTsafe opera como un "*sistema abierto relevador*" OSI y por tanto es factible un proceso de comunicación transparente, estandarizado y de fácil implementación.
- **Alta compatibilidad**: Una gran variedad de dispositivos ya desarrollados puede incorporar fácilmente IoTsafe, gracias a que sus módulos de seguridad y las aplicaciones de los dispositivos IoT emplean como interfaz *sockets* simples de red (API de *sockets*).

[104] "Cybersecurity and Infrastructure Security Agency", https://www.cisa.gov/

Las características de IoTsafe permiten dar respuesta también a la cuestión C7 planteada en el apartado 1.3:

C7: ¿Este esquema de seguridad podría llegar a ser compatible con algún sistema futuro de certificación más sencilla y automatizable?

IoTsafe contribuiría a resolver esta cuestión, ya que el desacoplo de seguridad de las aplicaciones además permite:

- Desarrollar módulos de seguridad estándar para las distintas arquitecturas *hardware* y diferentes niveles de seguridad, sin necesidad de tener en cuenta los aspectos de la aplicación IoT cuyas comunicaciones van a protegerse.
- Simplificar los procesos y costes de certificación y auditoría sin incurrir en una pérdida de conectividad en los ecosistemas IoT o en un aumento en la complejidad del desarrollo de los dispositivos y *software*. Además, los módulos de seguridad son reutilizables en dispositivos diferentes.
- Que las empresas desarrolladoras de tecnología IoT puedan delegar a terceros el desarrollo y mantenimiento de los módulos de seguridad, sin requerir una estrecha colaboración entre las partes.
- Reducir el tiempo y coste asociado al soporte post-venta de los dispositivos por incidencias de seguridad. Si existen vulnerabilidades o cambios en las certificaciones, las empresas especializadas en seguridad se encargarán de una forma más efectiva de proceder con las modificaciones y actualizaciones necesarias, de un modo similar al que utilizan las empresas que ofrecen *firewall* o antivirus.
- Estandarizar y regular la manera en la que dichas empresas especializadas de seguridad pueden gestionar la seguridad, sin incurrir en restricciones extra en los desarrollos del *software* IoT (obligación de utilizar determinados frameworks de desarrollo, etc.).
- Incrementar la confianza del consumidor en la tecnología IoT: las regulaciones son más fácilmente certificables externamente.
- Los tiempos de respuesta ante una incidencia de seguridad se pueden reducir drásticamente: no es necesario modificar las aplicaciones IoT de los dispositivos. Únicamente se debe actualizar el módulo de seguridad *stand-alone* IoTsafe, y esto puede automatizarse por las empresas especializadas en seguridad que gestionen los módulos de seguridad de los dispositivos IoT.

4. PRUEBAS Y RESULTADOS DEL ESQUEMA DE COMUNICACIONES SEGURAS IOTSAFE

Como se ha visto, IoTsafe propone utilizar elementos *software* de uso común en despliegues IoT. Esta clase de software se combina, para utilizarlo de una manera nueva que es necesario validar para determinar mejor las limitaciones que puedan surgir de esta innovación. Se consideró de interés evaluar el sistema propuesto en la sección 3.3, utilizando las versiones oficiales de los módulos software (*kernel* de Linux, Netfilter, SSH). Esta política que sigue IoTsafe permite que sea fácilmente implementable en una gran variedad de dispositivos, al no requerir de modificaciones *software* de relevancia: el software es de uso común y ya tiene soporte para múltiples arquitecturas y, además, en una gran variedad de despliegues IoT esos módulos software ya están presentes simplificando la implementación de IoTsafe. Obviamente, si se optimizaran esos módulos para este uso específico, se podrían obtener mejores resultados, pero como veremos, los obtenidos son buenos usando las implementaciones estándar.

Así, en este capítulo se presentan diversas pruebas experimentales del esquema de comunicación IoTsafe basado en el desacoplo de seguridad. Particularmente, el de una realización sencilla del mismo utilizando contextos de seguridad en entornos POSIX/Linux y el protocolo SSH. Este esquema basa su principio de funcionamiento en elementos *software* ampliamente testados en entornos productivos. En las diferentes secciones de este capítulo se realizan análisis detallados que permiten caracterizar sus ventajas y determinar sus limitaciones, a la hora de usarse como se plantea por IoTsafe. Asímismo, también se ofrecen soluciones a los principales problemas de escalabilidad encontrados.

A lo largo del capítulo también se realiza una comparación de este método de comunicación, basado en el desacople de la seguridad, frente al tradicional que emplea TLS/DTLS integrado en las aplicaciones. Por consiguiente, este capítulo dará respuesta detallada a la cuestión 8 (C8) planteada en la sección 1.3:

C8: ¿Qué efectos tendría la aplicación de este esquema de seguridad en los dispositivos IoT frente a las soluciones de comunicación segura tradicionales?

Los efectos que IoTsafe presenta en dichas comunicaciones se analizan en varios ámbitos. En la sección 4.1 se estudia el efecto que tiene IoTsafe en dispositivos IoT con *hardware* estándar y utilizando protocolos de comunicación estándar, tal y como presentamos en [136]. El análisis se efectua en un entorno de movilidad para evaluar el impacto en el rendimiento y la latencia.

La sección 4.2 realiza ese estudio en entornos estáticos, utilizando *hardware* de recursos limitados, pero manteniendo protocolos de alto nivel. Dichos análisis que presentamos en [108], se centran principalmente en el ancho de banda usado, *throughput* y consumo de energía.

La sección 4.3 extrae algunas conclusiones de las secciones anteriores y profundiza en el análisis, extendiéndolo al impacto de las comunicaciones proxificadas en movimiento que utilicen *hardware* y protocolos de recursos limitados. Estos resultados los publicamos en [137].

La sección 4.4 presenta un análisis detallado de las implicaciones que tiene en el *gateway*/servidor el esquema de comunicación presentado basado en proxificaciones, tal y como publicamos en [107]. Se analizan las principales limitaciones en términos de rendimiento y consumo de recursos, y se exploran soluciones que permitan la escalabilidad de este tipo de soluciones.

4.1. RENDIMIENTO DE LAS COMUNICACIONES PROXIFICADAS EN ENTORNOS ESTÁNDAR DE MOVILIDAD

La proxificación de las comunicaciones desde los dispositivos hacia los servidores puede introducir latencias no deseadas y degradaciones en el grado de conectividad del dispositivo (porcentaje de tiempo accesible) si éste se encuentra en movimiento en un entorno con una cobertura no óptima [138]. En este contexto, en una serie de pruebas publicadas en el artículo [136], estudiamos el impacto de este aspecto en un escenario real, en el que se proponen una serie de test en una prueba de campo empírica. Para ello se analizó el impacto de la proxificación del tráfico de aplicación y se mide la penalización de rendimiento en términos de *Round Trip Time* (RTT). El escenario propuesto (Fig. 4.1-1) se llevó a cabo en un vehículo móvil. Este vehículo seguía una ruta (Fig. 4.1-2) en la ciudad de Durango, DGO, México, en la que existen distintos niveles de señal de cobertura de la compañía de telefonía usada (Movistar México). El *hardware* empleado en la prueba consistió en un *player digital signage* basado en Linux Debian OS con una versión del *kernel* 3.10.48. La versión de OpenSSH utilizada para realizar las proxificaciones es la 6.6. El módem 3.5G+ utilizado era un Huawei E173 capaz de proporcionar un canal de bajada *High Speed Downlink Packet Access* (HSDPA) de hasta 7.2 Mbps y un canal de subida High Speed Uplink Packet Access (HSUPA) de hasta 2.1 Mbps.

Las pruebas consistieron en establecer un flujo de datos *background* para tratar de copar la capacidad del canal de subida utilizando la herramienta Iperf[105] con la que poder estimar el ancho de banda del canal. El cliente Iperf del *player* DS se conecta con un servidor Iperf ubicado en un servidor dedicado en Montreal, Canadá. Dicha conexión se produce a través de una proxificación por *port-forwarding* en una sesión SSH establecida que se mantiene a lo largo de todo el test.

[105] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP" https://iperf.fr/

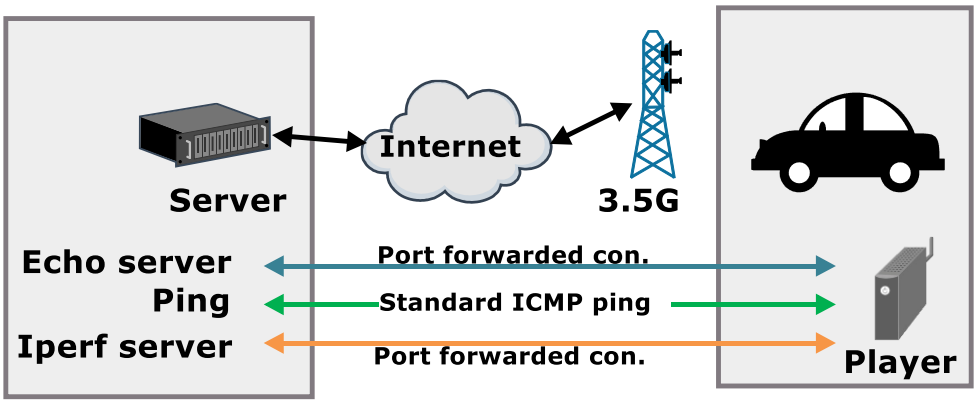


Fig. 4.1-1: Escenario de pruebas experimental en movimiento para medir la degradación causada por las comunicaciones proxificadas por port-forwarding de SSH en presencia de tráfico background (Iperf).

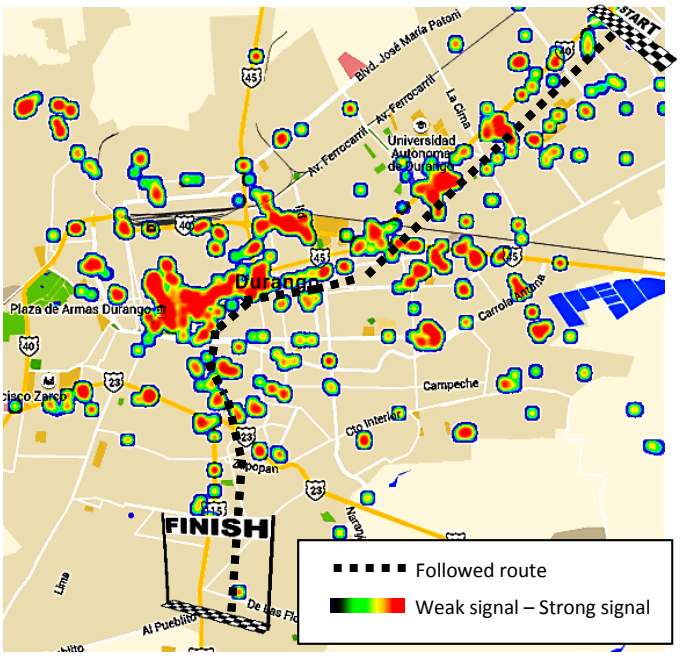


Fig. 4.1-2: Trayecto urbano seguido durante la prueba en la ciudad de Durango, DGO, México; superpuesto al mapa de nivel de señal de Movisar, proporcionado por OpenSignal[106] en 2015.

En paralelo, se establece una segunda sesión SSH desde el *player* DS con el servidor en Montreal, por la que se dispone una segunda proxificación que comunica con un servidor *echo*. Finalmente, se configura un *script* que establezca directamente comunicaciones ping periódicas con el servidor en Canadá mediante el protocolo ICMP.

El test evalúa el rendimiento de la comunicación proxificada frente a las comunicaciones no proxificadas, realizando medidas sencillas de RTT cuando existe un tráfico *background* importante desde el dispositivo hacia el servidor. En el *player* no se aplica ningún tipo de

[106] OpenSignal "Mobile Analytics and Insights", http://opensignal.com/

configuración QoS específica y opera el sistema de colas *pfifo_fast* [139] estándar en la mayoría de implementaciones Linux.

La Fig. 4.1-3 muestra el *throughput* de subida utilizado por Iperf, que sirve como indicador del ancho de banda del canal disponible, dado que es la única aplicación de uso intensivo del canal. Las otras dos comunicaciones en paralelo consisten en medidas periódicas del RTT que hacen un uso marginal del ancho de banda. Estos resultados se muestran en la Fig. 4.1-4.

En la ambas figuras también se aprecia el efecto de la proximidad a las antenas de la red de telefonía móvil y de los cambios en la calidad de la cobertura. Este efecto se traduce en los valores de RTT mostrados en la Fig. 4.1-4 y resumidos en la Tabla II, en los que se aprecia una leve degradación de un 1.7% de las comunicaciones en términos de RTT, debido a la proxificación de las mismas frente a la comunicación directa. Por otro lado, la conectividad de la comunicación proxificada apenas sufre degradación.

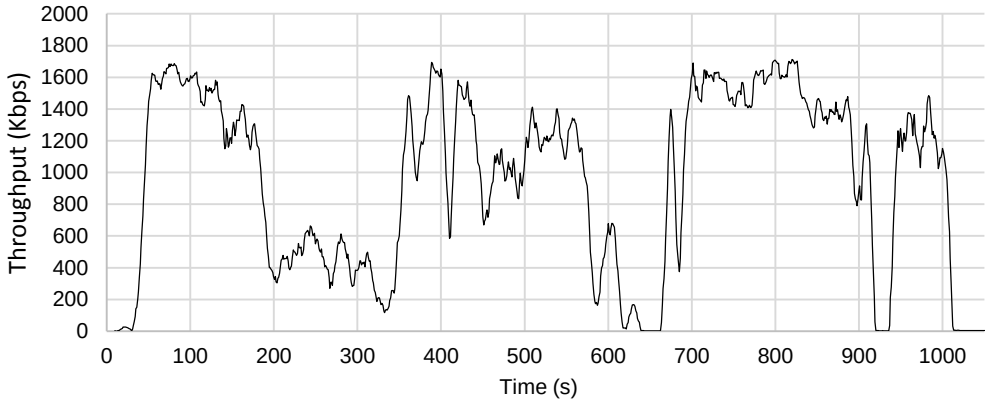


Fig. 4.1-3: Ancho de banda de subida medido desde el player en el vehículo en movimiento utilizando Iperf a través de una proxificación SSH con el servidor.

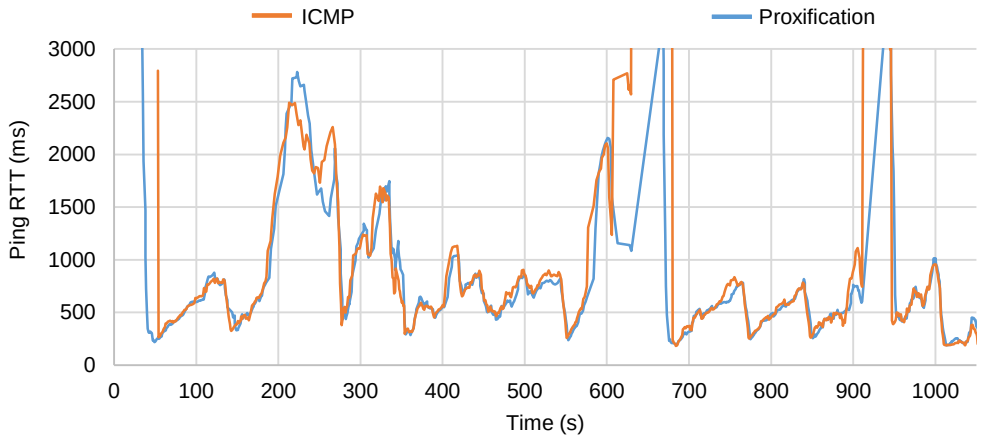


Fig. 4.1-4: Comparación de las latencias de una segunda proxificación en una sesión SSH diferente frente a la de los mensajes ICMP cuando Iperf se encuentra transmitiendo por la primera proxificación. Los valores de RTT superiores a 3 s. han sido recortados ya que se consideró este valor como umbral de una desconexión.

TABLA II
RESULTADOS COMPARATIVOS ENTRE ICMP Y LOS MENSAJES ECHO TRANSMITIDOS POR PORT-FORWARDING DE SSH

Métrica	*Echo* basado en ICMP	*Echo* basado en *Port Forwarding*
Conectividad	90%	89%
RTT Promedio	622 ms.	632 ms.

Se puede observar que la proxificación de las comunicaciones a través de OpenSSH apenas afecta a la calidad de las mismas en términos de RTT o de conectividad. La latencia promedio en presencia de tráfico *background* compartiendo una conexión PPP del enlace 3G resulta moderadamente alta (en torno a los 626 ms.), debido principalmente a las fluctuaciones de cobertura de 3G. Sin embargo, este tipo de escenarios puede resultar aceptable en comunicaciones breves IoT empleando protocolos que implementen una arquitectura de tipo *Representational State Transfer* (REST) [140]. Esto ocurre en algunas aplicaciones basadas en CoAP o HTTP, que no requieren valores reducidos de RTT para funcionar. El uso de protocolos RESTful como HTTP, también fomenta la interoperabilidad con sistemas de información externos de tipo ThingSpeak[107] mediante métodos básicos: GET, POST, PUT, DELETE.

4.2. ESTUDIO DE LAS COMUNICACIONES PROXIFICADAS DE PROTOCOLOS HTTP/HTTP2 CON *HARDWARE* PARA ENTORNOS LIMITADOS

Esta sección incluye la descripción de los entornos de pruebas que se prepararon para evaluar la factibilidad del esquema de seguridad propuesto utilizando los protocolos HTTP y HTTP/2. La seguridad en las comunicaciones que utilizan estos protocolos se proporciona mediante TLS y este esquema se comparará con el propuesto basado en SSH. El análisis pretende evaluar múltiples aspectos del sistema de comunicación propuesto frente a TLS, como la factibilidad del esquema conceptual en los dispositivos IoT y las características básicas necesarias para su implementación.

El análisis se centra particularmente en el rendimiento de los protocolos SSH frente a TLS. Estos pueden ser implementados en una gran variedad de lenguajes y siguiendo distintas metodologías. Algunos de los aspectos que influyen en el rendimiento de la implementación es el tipo de implementación de Netfilter o base de datos. También influye el grado de optimización de los scripts de sistemas empleados. Otros aspectos que también influyen en el rendimiento de la implementación son la configuración [141] o el tipo de implementación del framework de red [142] [143] o de la base de datos [144]. El comportamiento final del sistema también depende del grado de optimización de los scripts utilizados [145]. Sin embargo, el aspecto fundamental capaz de determinar la viabilidad del desarrollo de IoTsafe y su grado de escalabilidad radica en el protocolo de proxificación para las comunicaciones seguras que se emplée y la implementación que se haga

[107] ThingSpeak community, “ThingHTTP”: http://community.thingspeak.com/documentation/apps/thinghttp/

de él. Por ese motivo, el trabajo desarrollado tanto en esta sección como en la 4.3 y la 4.4 se centra en estudiar este aspecto principalmente.

A continuación, se presentan dos escenarios de pruebas. En el primero, se persigue caracterizar el rendimiento en dispositivos IoT que utilicen HTTP proxificando las comunicaciones de forma desacoplada de la seguridad por medio de SSH. Este procedimiento se compara con el método general de comunicación en el que la seguridad se integra en el protocolo de aplicación, en HTTPS (por medio de TLS). Esto se aplica también a HTTP/2. En el segundo escenario se persigue caracterizar dicho rendimiento en el servidor, para así poder dar respuesta a las cuestiones planteadas al inicio de la sección. Principalmente se persigue validar la escalabilidad y el consumo de RAM en los casos en los que existen múltiples comunicaciones simultaneas. Los escenarios de pruebas presentados se inspiran en el trabajo desarrollado como parte de la tutela de un proyecto de estancia en la Universidad Politécnica de Durango (Unipoli) a lo largo de 2016 [146]

En el primer escenario, representado en la Fig. 4.2-1, se utilizan dos dispositivos IoT capaces de soportar HTTP, TLS y SSH. Los elementos *hardware* usados son dos Raspberry Pi 3 modelo B V1.2 equipados con una interfaz 802.15.4 de Openlabs[108] (basados en la interfaz at86rf23 de Amtel).

La comparación entre TLS y SSH se realiza empleando dos versiones diferentes del protocolo HTTP: La 1.1 y la 2. Se compara HTTPS (HTTP con TLS) frente a HTTP protegido por medio de canales SSH seguros. En ambos casos se emplean algoritmos de seguridad estándar, aceptados y usados actualmente en numerosos casos reales. Además, determinadas suites de algoritmos TLS se pueden replicar en SSH a través de una configuración de sus algoritmos específica, lo que simplifica la comparación entre ambos casos.

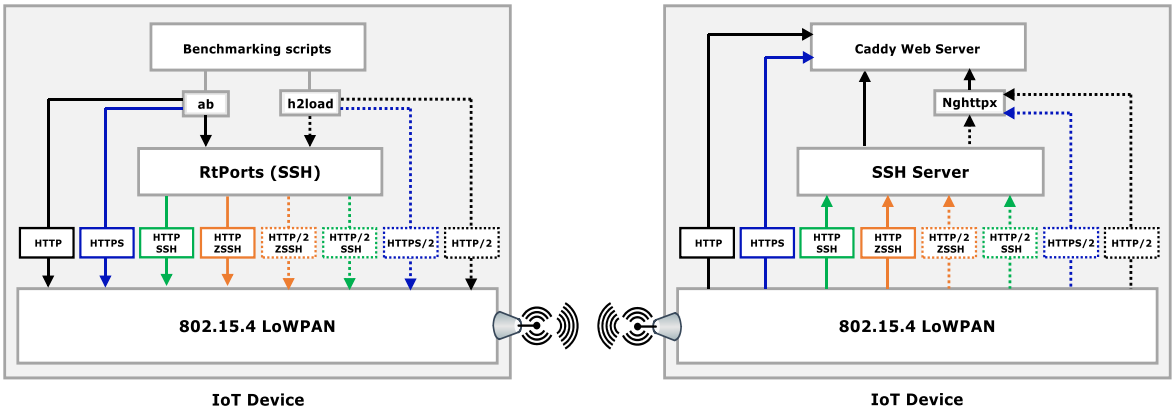


Fig. 4.2-1: Diagrama de pruebas de un dispositivo IoT para comparar el rendimiento en las comunicaciones entre SSH y TLS.

[108] "Openlabs Raspberry Pi 802.15.4 radio", https://openlabs.co/store/Raspberry-Pi-802.15.4-radio

El servicio HTTPS es proporcionado por Caddy[109], un servidor web portable que soporta TLS 1.2. Durante el proceso de testeos, se utiliza para asegurar las comunicaciones con la *suite* de protocolos 'TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384', clasificada como de clase B por la *National Security Agency* (NSA) [147]. Esta suite implementa el protocolo de intercambio de claves efímeras Diffie-Hellman basado en curvas elípticas (*Elliptic-curve Diffie-Hellman Ephemeral key Exchange*) utilizando el algoritmo *Elliptic-curve Digital Signature* (ECDSA). Junto a este, se emplea el cifrado de bloque AES-256 y el SSH-384 como código hash de autenticación de mensaje (HMAC). El certificado empleado por Caddy es autofirmado y, por tanto, no se efectua validación por la autoridad certificadora.

La versión de SSH se proporciona por OpenSSH y se encuentra configurada para utilizar el curve25519, sha256 como algoritmo ECDSA de intercambio de llaves, el aes256-gcm como algoritmo de cifrado de bloques y el hmac-512 como algoritmo HMAC para elaborar el código hash de autenticación de mensaje. Las funcionalidades de compresión de SSH son proporcionadas por la librería ZLIB [83] y se puede activar o desactivar por defecto editando el fichero de configuración ssh_config.

4.2.1. **Resultados de las pruebas de los dispositivos IoT**

En esta sección se presentan los resultados obtenidos en el primer escenario, que incluye dos dispositivos IoT. Las principales figuras de mérito empleadas son el ancho de banda y el tiempo de ejecución de los test. También se analiza el tiempo de computación y la eficiencia. A continuación, se contemplan los cuatro casos diferentes analizados, que conciernen a HTTP y a HTTP/2:

a) *HTTP:*** comunicaciones entre los dos dispositivos utilizando HTTP sin seguridad.

b) *HTTPS:*** HTTP con seguridad proporcionada por TLS1.2.

c) *HTTP SSH:*** HTTP con seguridad sobre SSH.

d) *HTTP ZSSH:*** HTTP con seguridad sobre SSH con compresión del *payload* por ZLIB.

Los test consisten en la realización de peticiones HTTP GET de ficheros de texto de diferentes tamaños con un rango de los 4 a los 262 Kbytes, simulando conexiones cortas y largas. Esta forma de proceder resulta de particular utilidad a la hora de comparar distintas situaciones que pueden darse en un entorno real, incluyendo el tráfico ordinario del funcionamiento de dispositivos IoT (comunicaciones cortas) junto con tráfico esporádico, pero importante, relativo a actualizaciones de *firmware* (transmisiones más largas) y comunicaciones más pesadas de servicios avanzados (multimedia, voz/datos, telemetría, etc.). Por cada caso se cursan 100 peticiones independientes de ficheros del mismo tamaño, pero de contenido aleatorio. Las 100 peticiones se realizan de forma secuencial, de modo que una petición no pueda comenzar sin que la anterior haya finalizado antes.

[109] "Caddy - The HTTP/2 Web Server with Fully Managed TLS", https://caddyserver.com/

El promedio de los resultados se calcula y presenta en diversas figuras que utilizan escalas no lineales en ambos ejes.

En los test relativos a HTTP se utiliza la herramienta de benchmarking ab[110] mientras que *hdload*[111] se emplea cuando las comunicaciones emplean HTTP/2. El servidor Caddy es proxificado por medio de *nghttpx*[112] para proporcionar soporte para conexiones HTTP/2 por medio del *token* h2c de actualización de la comunicación [148] que incluye dicho protocolo. El servidor Caddy proporciona soporte nativo para HTTPS/2, pero no para HTTP/2 y, por eso, se requiere de *nghttpx*.

La Fig. 4.2-2 presenta la eficiencia de las comunicaciones como la relación "*goodput* / *throughput*". El *goodput* es la métrica calculada por las herramientas de *benchmark* y representa la cantidad de información útil recibida (tamaño del archivo solicitado) en el tiempo que se ha tardado en recibir. El *troughput* corresponde al total de información enviada por la interfaz de red en el tiempo en el que dura dicho test. En el *overhead* se incluye también toda la información intercambiada requerida para asegurar las comunicaciones: paquetes de control de los protocolos, encabezados, retransmisiones, etc.

A este respecto, puede observarse que HTTPS presenta el peor comportamiento para cada tamaño de fichero, pero en particular con los pequeños. La razón que explica este rendimiento tan pobre se encuentra en la cantidad de *overhead* que HTTPS debe afrontar por cada petición: un intercambio y validación completo de certificados por cada una de las 100 peticiones GET.

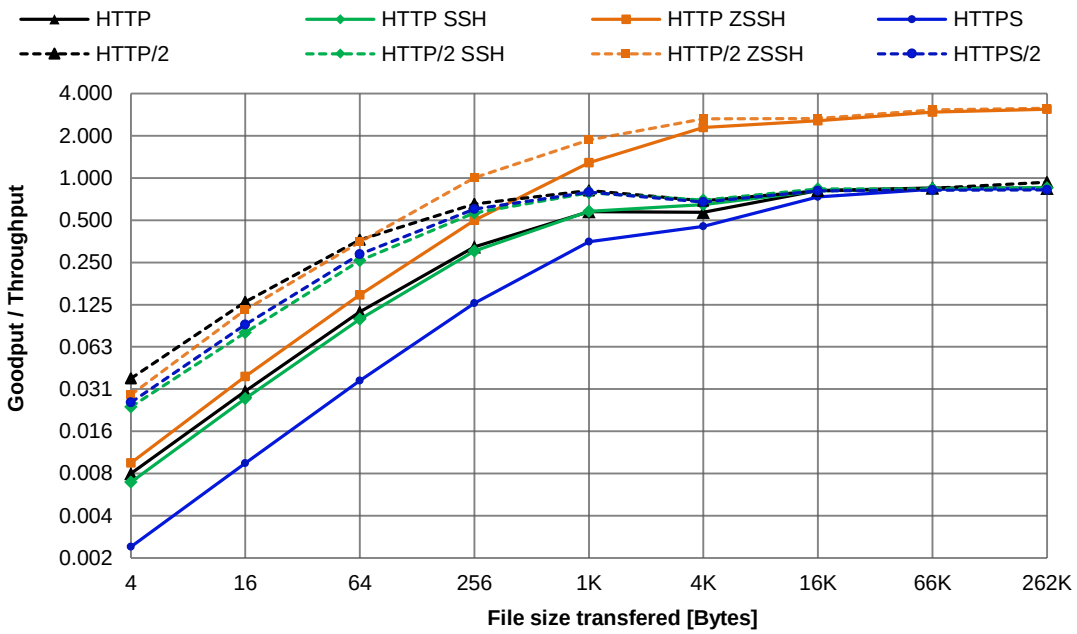


Fig. 4.2-2: Eficiencia en el uso del canal por cada caso analizado.

[110] "ab - Apache HTTP server benchmarking tool", https://httpd.apache.org/docs/2.4/programs/ab.html
[111] "h2load - HTTP/2 benchmarking tool - HOW-TO", https://nghttp2.org/documentation/h2load-howto.html
[112] "nghttpx - HTTP/2 proxy - HOW-TO", https://nghttp2.org/documentation/nghttpx-howto.html

Sin embargo, esto último no es necesario en el caso HTTP SSH, ya que SSH mantiene un canal de comunicación abierto, así que no requiere dichas verificaciones más que en su establecimiento: todas las peticiones GET basadas en HTTP posteriores se pueden procesar sin requerir medidas de seguridad adicionales. Como ejemplo, se puede observar la métrica de eficiencia relativa a las peticiones GET de 256 bytes: en el caso de HTTP SSH, la eficiencia es de 0,3 mientras que únicamente lo es de 0,13 para HTTPS (2,3 veces menos). El impacto de este fenómeno es inferior conforme el *payload* se incrementa, pero en todo caso, HTTPS siempre se comportará peor. Algunas razones que pueden explicar esta diferencia son las siguientes:

- Las conexiones HTTP por medio de SSH (HTTP SSH) se manejan a través de canales de la sesión SSH. Es posible que el *payload* existente en múltiples paquetes TCP acabe siendo transmitido de forma más compacta por SSH; si bien esto último no parece suficiente para explicar una mejora tan radical.

- Otro motivo de relevancia es la forma de funcionar la comunicación basada de HTTP asegurada con SSH. La aplicación que quiere comunicarse siguiendo esta estrategia debe conectarse localmente al cliente SSH. En el servidor, el servidor SSH se conecta a destino. Esto ocurre con cada petición GET, y por tanto los paquetes TCP requeridos para el establecimiento de conexión, únicamente circulan a nivel local dentro de las máquinas. Esto evita que dichos paquetes tengan que transmitirse a través de la interfaz 802.15.4, ahorrando al menos un tiempo equivalente a 3 RTTs por cada petición GET.

- Finalmente, se encuentra el motivo de mayor peso: el evitar que con cada petición GET se tengan que verificar y validar los certificados derivados de la comunicación asegurada por TLS. A Este respecto, el protocolo TLS incluye en su definición una funcionalidad denominada *keep_alive* [149] que podría aprovecharse precisamente para mitigar esta problemática. Sin embargo, no puede considerarse una funcionalidad transparente, ya que requiere que una petición *heartbeat* se codifique inicialmente desde la aplicación y posteriormente se envíe (la aplicación debe programarse con la idea de utilizar el *keep_alive*).

- En la versión TLS 1.3, la funcionalidad del *keep_alive* se sustituye por la capacidad de retomar una sesión previamente establecida [150]. Aunque se han detectado posibles problemas de privacidad relacionados con este procedimiento [151], el impacto en IoT podría ser mucho menor que el experimentado en los exploradores web convencionales. Sin embargo, la funcionalidad de retomar la sesión requiere una programación en la aplicación IoT que, al igual que en el caso de *keep_alive*, vuelve a quebrar la transparencia de la capa de transporte segura con la de aplicación.

En todo caso, el margen de mejora se incrementa todavía más en el caso *HTTP SSH Compressed*, en donde se utilizan las prestaciones de compresión existentes en SSH y se

aprovecha la alta compresión del *payload*: los datagramas HTTP y su propio *payload* utilizan una semántica textual de alta compresión [152].

Por otro lado, cabe destacar la mejora técnica que supone el uso de HTTP/2. Este protocolo incorpora algunas de las características que ya proporciona SSH, y esto se traduce en una mejora sustancial del rendimiento. La herramienta de *benchmark* utilizada, *hdload,* hace uso de las prestaciones de *streaming* de HTTP/2, que permiten a este protocolo reutilizar las conexiones TCP, y por extensión evitar la operación de validación de certificados por cada petición. Este hecho, junto con la reducción de tamaño de los encabezados (al sustituir su formato "modo texto" por código binario comprimido), optimiza de forma sustancial todas sus transmisiones. Sin embargo, la alternativa de seguridad desacoplada por medio de SSH con compresión (extensible también al *payload*) todavía mejora aún más las métricas alcanzadas por HTTP/2.

En la Fig. 4.2-3 el *goodput* conseguido se compara entre protocolos. Este gráfico es congruente con el anterior, mostrando de nuevo una mejora sustancial en el rendimiento en las comunicaciones aseguradas con SSH frente a las otras alternativas. Cuando se llevan a cabo los test con HTTP/2, el incremento en el rendimiento ocurre únicamente si las características de compresión se activan. Este resultado mejora conforme aumenta el tamaño del fichero solicitado.

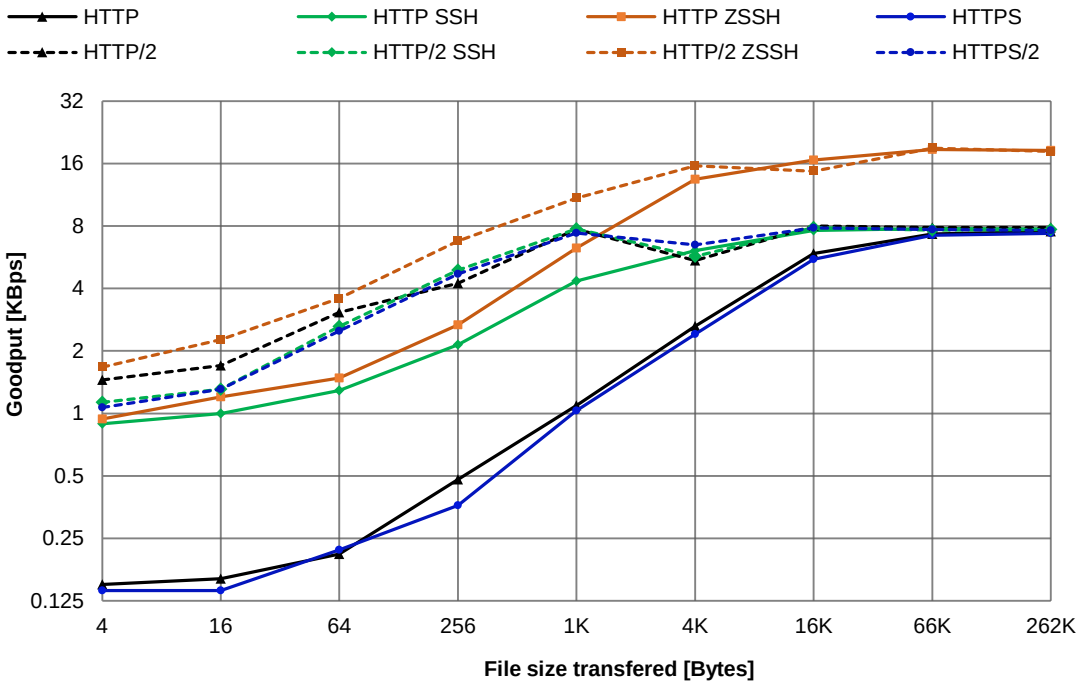


Fig. 4.2-3: ***Comparación del Goodput entre los casos analizados. Un overhead mayor no implica que el goodput sea menor necesariamente: el protocolo puede optimizar lo suficiente la comunicación***

En la Fig. 4.2-4 se muestra el impacto de usar la seguridad en términos de consumo de energía consumida por la CPU. Las medidas toman como referencia la energía utilizada para el caso HTTP. Dichos consumos de energía no son medidos directamente, sino extrapolados a partir del tiempo de CPU utilizado en cada caso *[153]*. Los tiempos de CPU empleados se obtienen utilizando la herramienta "*time*"[113] de Linux en los programas de *benchmarking,* y también en SSH cuando se utilizan las proxificaciones.

Los resultados obtenidos son coherentes con las anteriores gráficas y reflejan la carga extra en CPU que requiere el protocolo HTTPS. HTTPS/2 muestra una mejora notable gracias a las prestaciones mencionadas anteriormente. SSH obtiene peores resultados que HTTPS/2 en este caso con payload menores, debido en parte a la configuración del escenario de pruebas presentado para las medidas. El comando SSH se lanza para cada serie de 100 tests y la ejecución del mismo se computa como carga de CPU. Esto penaliza notablemente el resultado, debido a que la serie de tests con *payload* reducidos termina muy rápidamente, suponiendo esta parte una carga computacional muy baja comparada con la ejecución de SSH. Este fenómeno se minimiza cuando el *payload* aumenta, ya que el tiempo de ejecución total de la serie de 100 tests es mayor, y la ejecución del comando SSH comienza a tener menor peso. Cuando los payload son mayores (66 y 262 Kbytes), el comportamiento de SSH supera ligeramente al de HTTPS/2, pero la mejora más sustancial se obtiene a raíz de la habilidad de SSH de efetctuar compresión. Si dichas capacidades se habilitan, SSH mejora el comportamiento de HTTPS/2 a partir de peticiones de 16 Kbytes, debido a que resulta más efectivo en términos computacionales comprimir y cifrar menos datos, que únicamente cifrar (gracias a la elevada tasa de compresión

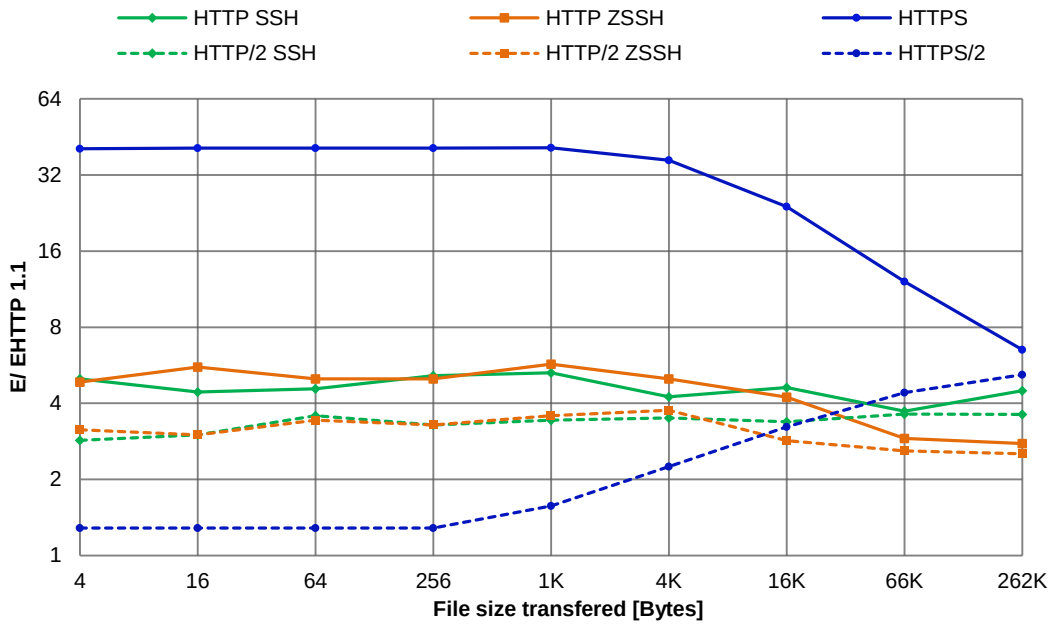


Fig. 4.2-4: Comparativa del consumo de energía de CPU en cada caso tomando la requerida por HTTP como referencia.

[113] "time - time a simple command or give resource usage" *Linux man page*, https://linux.die.net/man/1/time

del *payload*). Estos resultados indican que SSH puede comportarse mejor si efectua establecimientos de sesión persistentes o más duraderos en el tiempo (no sólo hasta las 100 conexiones).

En la Fig. 4.2-5, se muestra el impacto en el consumo de energía para cada caso, relativo al uso de la interfaz inalámbrica. En esta figura, como en la anterior, se muestra un incremento en el ahorro de energía cuando se utiliza SSH para asegurar las comunicaciones HTTP. Sin embargo, para HTTP/2 se produce una regresión cuando las peticiones se realizan en archivos pequeños. El principal motivo de este efecto radica en que el *payload* total, al ser muy escaso (pese a considerarse las 100 peticiones), es comparable con el *overhead* total requerido para establecer inicialmente la sesión SSH junto con el *overhead* inherente al emplear SSH: los interfaces de red deben transmitir más información y por tanto, pasan más tiempo encendidos. Este efecto, sin embargo, desaparece cuando el *payload* aumenta lo suficiente.

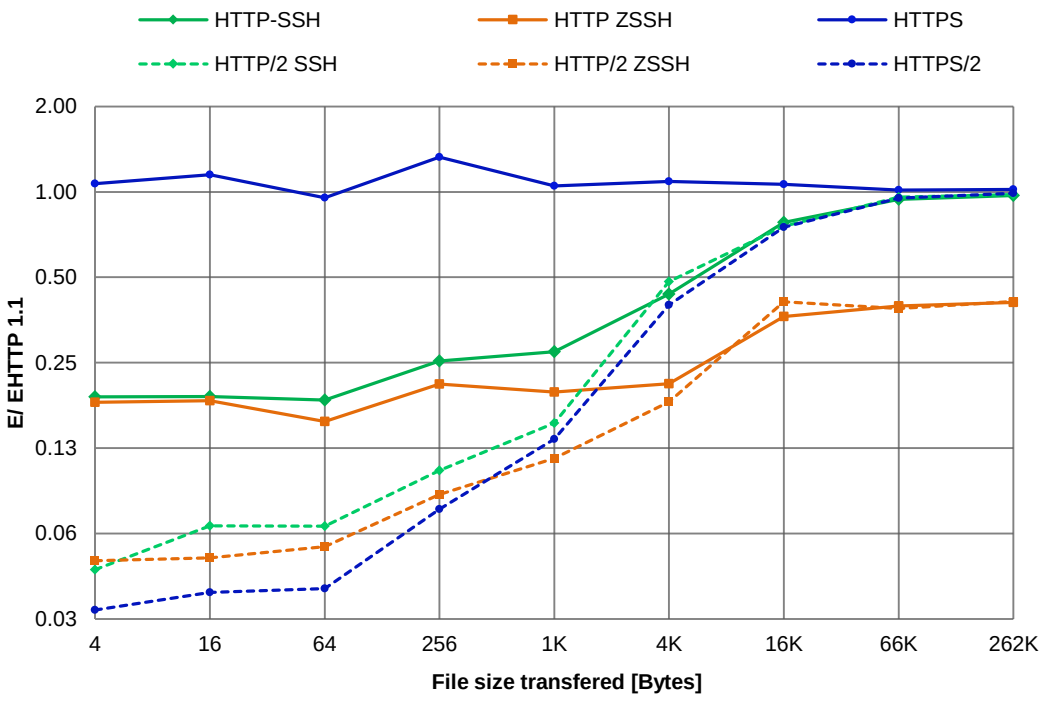


Fig. 4.2-5: Comparativa del consumo de energía de la interfaz wireless en cada caso tomando la requerida por HTTP como referencia.

4.2.2. **Análisis de los resultados**

Los resultados de las pruebas llevadas a cabo para IoTsafe demuestran que su esquema de comunicación es adecuado en términos de portabilidad y facilidad de actualización, una vez integrado en los dispositivos basados en HTTP o HTTP/2. Los test realizados comprenden dispositivos IoT que utilizan canales con prestaciones reducidas de comunicación de tipo 802.15.4, con MTU y ancho de banda limitados, propiciando situaciones de alto *overhead* y fragmentación. Dichos resultados muestran una mejora en términos de ahorro de energía y ancho de banda con

respecto a técnicas basadas en HTTP usando TLS. Respecto a HTTP/2 se consiguen también mejores cifras en el ancho de banda efectivo, a cambio de un incremento en el consumo de energía, pero únicamente cuando se transmiten archivos muy pequeños.

El esquema propuesto, basado en comunicaciones por proxificación por SSH, permite que HTTP y HTTP/2 funcionen con nuevas interfaces con restricciones en su MTU y ancho de banda. Esto se consigue gracias al esquema de proxificación que proporciona una seguridad independiente de la aplicación y de las librerías del *firmware* existentes en el *software* IoT.

Los resultados muestran cómo IoTsafe resulta una alternativa viable para diferentes proyectos IoT. Dado que se fundamenta en el uso de protocolos y tecnología Linux de largo recorrido y soporte, puede aprovechar sus funcionalidades sin incurrir en errores ocultos graves que afecten a medio y largo plazo. Este esquema de comunicación permite que aplicaciones RESTful muy simples puedan diseñarse sin tener que preocuparse por los aspectos complejos de seguridad.

4.3. ESTUDIO DE LAS COMUNICACIONES PROXIFICADAS DE COAP UTILIZANDO *HARDWARE* PARA ENTORNOS LIMITADOS

Para proseguir con el proceso de validación de la tecnología presentada por IoTsafe, se presentan una serie de nuevos test orientados esta vez a entornos que empleen protocolos de aplicación específicos para dispositivos con recursos limitados. Los resultados, publicados en [137] se basan en la utilización del protocolo de aplicación CoAP [154], que utiliza UDP como protocolo de transporte. Dado que la solución estándar planteada por IoTsafe únicamente ofrece soporte a aplicaciones basadas en TCP al emplear SSH como protocolo de proxificación, se utilizará un sencillo *middleware* para adaptar el tráfico UDP a TCP y viceversa. El estudio del impacto en el rendimiento de la solución de seguridad propuesta incluirá también la posible penalización que suponga emplear este *middleware*.

Dicho estudio comparativo utilizará peticiones GET a un *gateway*/*server* al que se solicita un recurso. Tanto el servidor como el cliente que soportan el protocolo CoAP están diseñados aprovechando el *framework* Californium[114]. El recurso que se ofrece es de carácter dinámico y basado en texto aleatorio (palabras en latín consideradas texto de tipo *Lorem Ipsum*). Las figuras de mérito que se pretende evaluar en estos test son el *overhead*, la ratio de pérdida de tramas 802.15.4 y el tiempo requerido para realizar cada una de dichas transferencias (T_{req}). Estas medidas están planteadas con el objetivo de caracterizar el impacto del desacoplo de seguridad en las comunicaciones en las que existen pérdidas de tramas por interferencias de otras transmisiones diferentes fuera de nuestro control. El efecto de la fragmentación y dependencia

[114] Eclipse Californium™, https://www.eclipse.org/californium/

con el *payload* se omiten dado, que se analizaron en profundidad en experimentos previos en la sección 4.2 y son perfectamente extrapolables a este caso [107].

Las series de test se efectúan en dos escenarios distintos: el primero presenta una ausencia prácticamente total de interferencias en el canal de transmisión; y el segundo sufre pérdidas moderadas de aproximadamente un 1% de las tramas 802.15.4 transportando datagramas UDP. El procedimiento seguido por los test en el caso del escenario con pérdidas, está diseñado particularmente para que los efectos del canal afecten estadísticamente de forma equivalente a cada una de las configuraciones de los protocolos analizados (utilizaremos el término *caso* para referirnos a cada una de las cuatro configuraciones). Las pruebas pertinentes que demuestran la validez de esta afirmación se presentan posteriormente en la lectura e interpretación de los resultados.

Los test se efectúan mediante 100 conjuntos de 100 peticiones GET por cada *caso*, que se llevan a cabo de forma intercalada. Cada una de las 100 peticiones GET de cada *caso* se realiza en serie, es decir, una a continuación de la otra cuando la primera termina. Tras esto, se procede con el siguiente *caso* de la misma manera. Una evaluación consistente en 100 peticiones GET en serie por cada *caso* constituye una iteración, y el conjunto de los test propuestos consiste en 100 iteraciones en total. Cada petición GET solicita un recurso de texto de 1024 bytes en total. Este tipo de recurso se ofrece en forma de texto aleatorio para cada petición. Como resumen, en total tenemos cuatro *casos*, que se examinan en dos escenarios.

El conjunto de test de 100 iteraciones se realiza de la misma manera en los dos escenarios. La serialización en cada iteración de las 100 peticiones GET en cada *caso* persigue evitar perjudicar al *caso* que utiliza DTLS (CoAPs). Tradicionalmente, DTLS no puede reutilizar la configuración establecida en las comunicaciones, teniendo que realizar todo el proceso de securización por cada petición GET. Sin embargo, si el cliente se configura de manera que éste pueda serializar las peticiones en el mismo establecimiento de comunicación, las configuraciones criptográficas sirven para las 100 peticiones GET en serie, evitando el proceso de establecimiento de canal seguro, validación de certificados, intercambio de llaves, etc. Este proceso no sería necesario en los *casos* basados en SSH, ya que debe existir una sesión previamente abierta y que se mantiene así de forma persistente. Por consiguiente, para evitar penalizar excesivamente a DTLS y poder estudiar otros posibles efectos, las series de test se plantean de manera que las 100 peticiones se realicen seguidas para que, de ese modo, DTLS evite tener que realizar el establecimiento criptográfico en cada petición.

Los cuatro *casos* que se consideran en cada escenario son los siguientes (Fig. 4.3-1):

- **CoAP**: Configuración insegura cliente/servidor utilizando aplicaciones basadas en CoAP. Este *caso* se plantea para obtener métricas que sirvan de referencia para los demás *casos*
- **CoAP SSH**: Configuración segura cliente/servidor utilizando aplicaciones basadas en CoAP protegidas con SSH. Este *caso* utiliza comunicaciones inseguras idénticas a las del primero, pero protegiéndolas de forma transparente por SSH mediante proxificación. Dado que SSH no permite proxificar flujos basados en UDP, se utiliza un *middleware* sencillo a partir del comando de Linux *socat.*
- **CoAPs**: Configuración segura cliente/servidor utilizando aplicaciones basadas en CoAPs. Este *caso* utiliza una configuración estándar CoAP que protege las comunicaciones por medio de DTLS.
- **CoAP SSH_C**: Configuración segura cliente/servidor utilizando aplicaciones basadas en CoAP protegidas con SSH con las prestaciones de compresión activadas. Este *caso* es idéntico al segundo (CoAP SSH), con la salvedad de que se habilita la compresión mediante la librería ZLIB [83] a los flujos de comunicaciones que se utilicen en los canales proxificados.

La configuración del testbed presentado en la Fig. 4.3-2, común para ambos escenarios, comprende un dispositivo actuando como cliente y otro como *gateway*/servidor. Ambos dispositivos son Raspberry Pi 3, modelo B V1.2, equipados con una interfaz 802.15.4 de Openlabs basada en el chip at86rf233 de Atmel. La comunicación entre ambas Raspberries se efectúa sobre una WPAN. El primer escenario utiliza el canal 26 (2480 MHz), para tratar de evitar las interferencias 802.11. El segundo escenario utiliza el canal 22 (2460 MHz), para evaluar un canal con pérdidas (algo habitual en escenarios IoT), debidas a la existencia de interferencias del canal 11 de 802.11. En nuestro caso, existen 34 redes WiFi en ese canal, con unas intensidades en el rango de -59 a los -75 dBm.

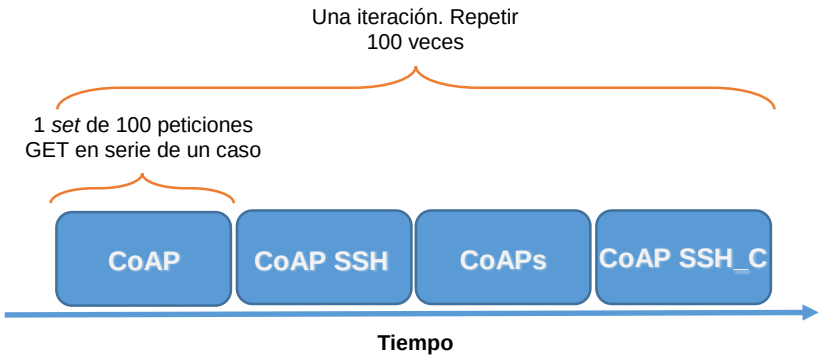


Fig. 4.3-1: Organización de la secuencia de peticiones GET entre los casos a lo largo del test para cada escenario.

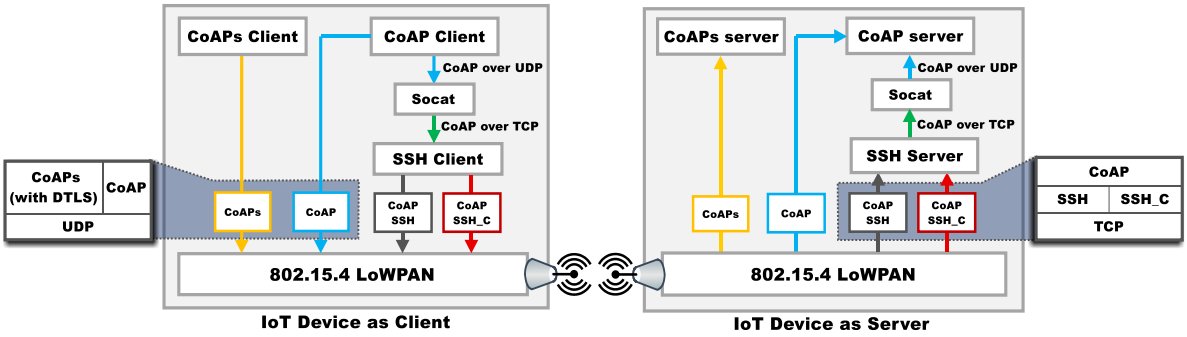


Fig. 4.3-2: Diagrama de las pruebas de rendimiento realizadas con CoAP para comparar SSH y DTLS.

Ambas interfaces están configuradas a una potencia de emisión de -2 dBm y colocadas a un metro de distancia una de otra. Se configura una interfaz virtual LoWPAN en ambos dispositivos para fragmentar los paquetes cuando se requiera. IPv6 utiliza una MTU mínima de 1280 bytes, mientras que la MTU mínima de 802.15.4 es de 127 bytes (el trabajar con una fragmentación LoWPAN implica que sólo quedan 96 bytes de *payload* efectivos). El *kernel* de Linux utilizado es el 4.7.4, que incluye una versión estable de la implementación LoWPAN de IEEE 802.15.4. La tasa de transferencia de estos dispositivos se configura a 250 Kbps, la máxima especificada en el estándar.

La implementación basada en CoAPs utiliza el protocolo DTLS 1.2 y el algoritmo 'TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256', uno de los recomendados en la especificación conocida como la *Extensible Authentication Protocol-Transport Layer Security* (EAP-TLS) *Approved Algorithms* [155]. Esta versión implementa el algoritmo de intercambio de claves temporales *Elliptic-Curve Diffie-Hellman,* junto con el algoritmo de firma *Elliptic-Curve Digital Signature Algorithm* (ECDSA), empleando un bloque de cifrado AES-128 y una configuración SHA-256 para el hash del código de autenticación de mensaje (HMAC).

La implementación de SSH utilizada es OpenSSH, configurada para trabajar con unos algoritmos similares, aunque más seguros, a los proporcionados por DTLS: SSH utiliza como algoritmo de intercambio de claves el ECDSA curve25519-sha256, Para el cifrado de bloques se utiliza el aes256-gcm y para HMAC el hmac-512. Las características de compresión de SSH son proporcionadas por ZLIB y se pueden activar por defecto editando el fichero ssh_config. Los parámetros TCP utilizados son los estándares en la distribución Raspbian, y los valores de CoAP utilizados por defecto se adaptan ligeramente para que el protocolo pueda soportar los test. Los únicos valores modificados son los siguientes: *blockwise_status_lifetime*, *max_resource_body_size* and *max_peer_inactivity_period*.

4.3.1. Caracterización del rendimiento de CoAP bajo comunicaciones proxificadas en entornos con pérdidas

La comparación entre DTLS y SSH se efectúa ejecutando los test directamente sobre CoAPs (que utiliza DTLS) y CoAP sobre SSH. El *caso* de CoAP (sin seguridad) se evalúa de igual modo, para disponer de una referencia. El servidor ofrece el recurso de texto a través de CoAP o CoAPs, tal y como se explica en la sección anterior, puesto que este tipo de formato resulta muy común en entornos IoT basados en CoAP; los *payload* suelen consistir en texto con descripciones semánticas [152].

Aunque este recurso no incluye código binario ejecutable como *payload* para las pruebas, se asume que sería igualmente comprimible, debido a las características de compresibilidad también presentes en este tipo de código binario [156]. Los tests se plantean para medir diferentes características del tráfico:

- **T_{req}:** Es el tiempo requerido para efectuar 100 peticiones GET consecutivas de 1024 bytes de texto aleatorio cada una. El efecto que tendría emplear otros valores diferentes de *payload* ya se ha estudiado extensamente en secciones anteriores del documento (las pruebas de HTTP/HTTPS en la sección 4.2), en donde se estudió el efecto que la variabilidad del tamaño del *payload* tiene sobre la fragmentación de las tramas 802.15.4. Se determinó que a mayor valor de *payload*, los efectos de compresibilidad mejoran el comportamiento de SSH con compresión frente a todas las demás opciones. Además, el hecho de mantener la sesión abierta vía SSH evita tráfico innecesario que se produce en TLS/DTLS. Por consiguiente, se opta por elegir un valor de *payload* de 1024 bytes, lo suficientemente grande para generar fragmentación, pero sin perjudicar en exceso a DTLS (cuanto más grande, mejor para SSH, sección 4.2.2).
- ***Overhead*:** Consideramos como *overhead* en la serie de test a la información extra recibida por la interfaz de red, más allá de los datos de la petición (100 peticiones GET de 1024 bytes cada una por *caso* en una iteración). El resto de información recibida corresponde a la utilizada por protocolos de comunicación, retransmisiones, etc. y se considera como *overhead*.
- **Error *rate*:** Porcentaje de tramas perdidas (tanto enviadas como recibidas) en relación al total de tramas cursadas (por *caso* e iteración).
- **Rendimiento ΔP:** En los test se entiende por mejora del rendimiento porcentual la diferencia porcentual del T_{req} del *caso* concreto frente al T_{req} de CoAPs.

4.3.1.1. Escenario sin pérdidas

En las Fig. 4.3-3 y Fig. 4.3-4 se presentan los resultados de los test en unas circunstancias de prácticamente ausencia de pérdidas. Este escenario es estático, con únicamente dos dispositivos 802.15.4: un emisor y un receptor, utilizando un canal libre de interferencias de

transmisiones 802.11 y, por ende, se asume que la suposición de ausencia de pérdidas puede considerarse válida.

En estos resultados se puede observar el efecto del sistema de control de congestión estándar en la distribución Raspbian (control de congestión Reno) en los casos que utilizan SSH. Mientras que las transferencias efectuadas mediante UDP (CoAP y CoAPs) no sufren más que alguna pérdida anecdótica de tramas, las comunicaciones basadas en TCP (CoAP_SSH y CoAP_SSH_C) siempre incluyen porcentajes nada despreciables de pérdidas. Este algoritmo, Reno, trata de sacar el máximo partido al ancho de banda del canal, generando pérdidas de tramas. Por el otro lado, el algoritmo más conservador de CoAP, quizás no explote al máximo la capacidad del canal, pero no genera pérdidas.

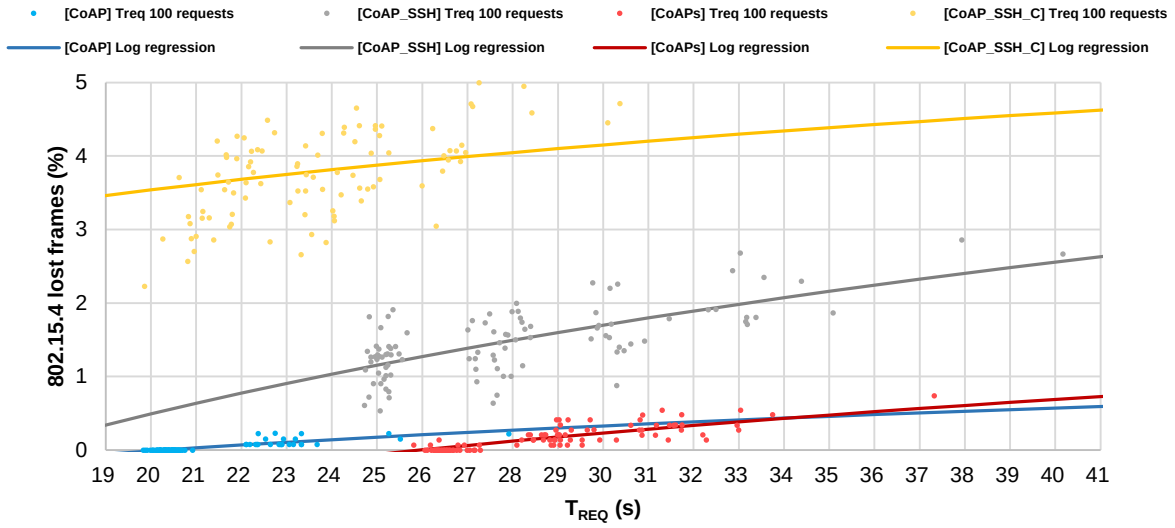


Fig. 4.3-3: Dispersión de T_{req} en los conjuntos de 100 peticiones CoAP GET de 1024 bytes de payload cada una con un promedio de LQI causante de la pérdida de ~0,1% de tramas 802.15.4 de tráfico UDP's.

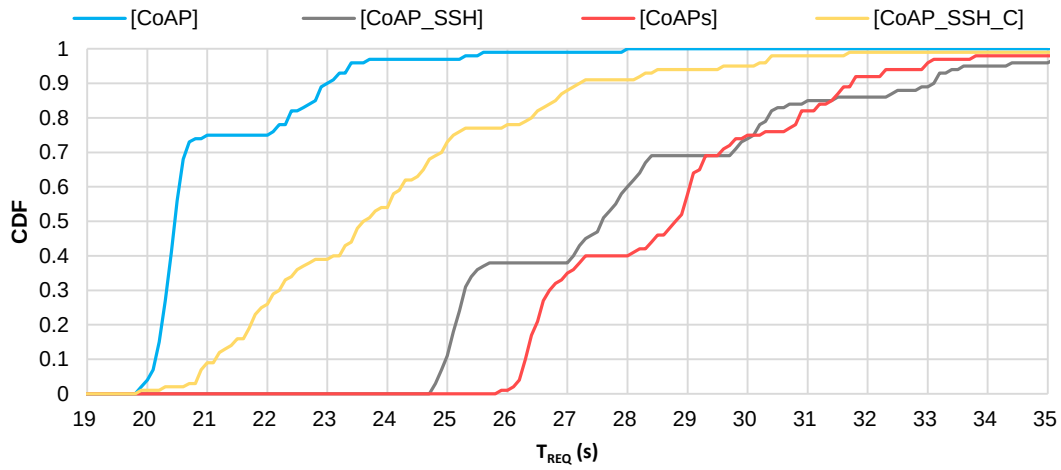


Fig. 4.3-4: CDF de T_{req} de 100 peticiones GET de 1024 bytes de payload cada una con un LQI promedio causante de la pérdida de ~0.1% de tramas 802.15.1 relativas al tráfico UDP.

Estos resultados muestran cómo la tasa de pérdidas de tramas derivada de las transmisiones en los *casos* CoAP_SSH y CoAP_SSH_C no sirven para estimar realmente la capacidad del canal (se producen péridas de tramas sin interferencias). Por el contrario, las métricas de pérdidas de tramas de los *casos* CoAP y CoAPs, no se ven afectadas por el algoritmo de control de congestión (no se producen pérdidas de tramas). Por consiguiente, la métrica de tramas perdidas de CoAP y CoAPs sí puede tenerse en consideración a la hora de estimar la capacidad real del canal en caso de interferencias.

Un resumen de estos resultados se presenta en la Tabla III. De ellos se deduce que, aunque COAP_SSH sufre un incremento del 38% en su *overhead* sobre CoAPs, su rendimiento mejora al último ligeramente por un 2,29% gracias a un mejor uso del canal. Si las funcionalidades de compresión se encuentran activadas, la mejora en rendimiento de CoAP_SSH_C frente a CoAPs alcanza el 14,26%.

Durante todos los test, las posibles interferencias sufridas se tratan como ruido [157]. Ese es el comportamiento estándar configurado en el chip (*cca_mode* = 1) [115] para todas las señales interferentes por encima de -77 dBm (*cca_ed_level* = -77). Una estrategia aceptada para tratar de modelar estas señales interferentes consiste en interpretarlas como ruido causante de desvanecimientos del canal o *channel fading* [158].

En enlaces 802.15.4, el *Link Quality Indicator*[9] (LQI) se presenta como una métrica más adecuada para medir la calidad del canal real, frente a otros indicadores tradicionales como el *Received Signal Strength Indicator* (RSSI). Esto se debe principalmente a que el efecto que tienen interferencias de banda estrecha en el ancho de banda del canal puede causar *fading* en las transmisiones, pese a mantenerse el nivel de RSSI; o incluso llegar a incrementarse por acción de dichas interferencias. Por ello, el LQI puede calcularse mejor considerando también métricas relativas a errores del chip [159]. De igual modo, las tramas perdidas 802.15.4 podrían servir como indicador válido del LQI si no hubiese otra causa más que afectase a su variabilidad, como en los *casos* CoAP y CoAPs previamente presentados.

TABLA III
RESUMEN DE LOS RESULTADOS DEL 1ER ESCENARIO

Caso	Error de trama promedio (%)	T_{req} promedio (s)	ΔP* (%)	Overhead promedio (%)
CoAP	0.03	21.07	27.00	11.92
CoAPs	0.16	28.86	0.00	20.23
CoAP_SSH	1.49	28.20	2.29	58.92
CoAP_SSH_C	3.82	24.74	14.26	-8.14

* $\Delta P\ (\%) = 100*(T_{req}\_CoAPs - T_{req})/T_{req}\_CoAPs$

[115] Atmel, "AT86RF233 preliminary datasheet", 2014, http://ww1.microchip.com/downloads/en/devicedoc/atmel-8351-mcu_wireless-at86rf233_datasheet.pdf

4.3.1.2. Escenario con pérdidas

La Fig. 4.3-5 presenta los resultados de los test en un escenario con pérdidas. Se trata de un canal real con interferencias 802.11, donde las fuentes de dichas interferencias escapan a nuestro control.

La naturaleza de otros protocolos que están causando interferencias con las comunicaciones 802.15.4 ha sido estudiada en profundidad en otros artículos utilizando cámaras anecóicas [160]. Sin embargo, en los test que se abordan en esta sección, y que han sido publicados en [137] se carece de tal grado de control: se presenta un escenario realista con interferencias derivadas de comunicaciones reales 802.11, que resulta difícil de modelar con exactitud. Todo esto representa un reto que este trabajo debe superar para poder desarrollar los test en los cuatro *casos,* en un canal que mantenga unas condiciones equivalentes para evitar obtener conclusiones sesgadas.

Para abordar esta situación, el efecto de las interferencias en el canal se ha analizado *caso* por *caso,* comparando la correlación del T_{req} de cada test, con el porcentaje de tramas perdidas 802.15.4. Estos resultados se han mostrado en la Fig. 4.3-5 como un gráfico de dispersión, junto con la regresión logarítmica correspondiente para cada serie de resultados. El comportamiento de los algoritmos de control de congestión (tanto el de TCP Reno [161] como el estándar de CoAP [162]) tienen un comportamiento logarítmico, debido a que los tiempos de *backoff* crecen exponencialmente: si el canal presenta problemas y para evitar profundizar en la congestión, dichos algoritmos retrasan la emisión de los paquetes[116] de forma transparente a la aplicación, con espacios de tiempo más largos tras cada error consecutivo detectado (datagrama perdido).

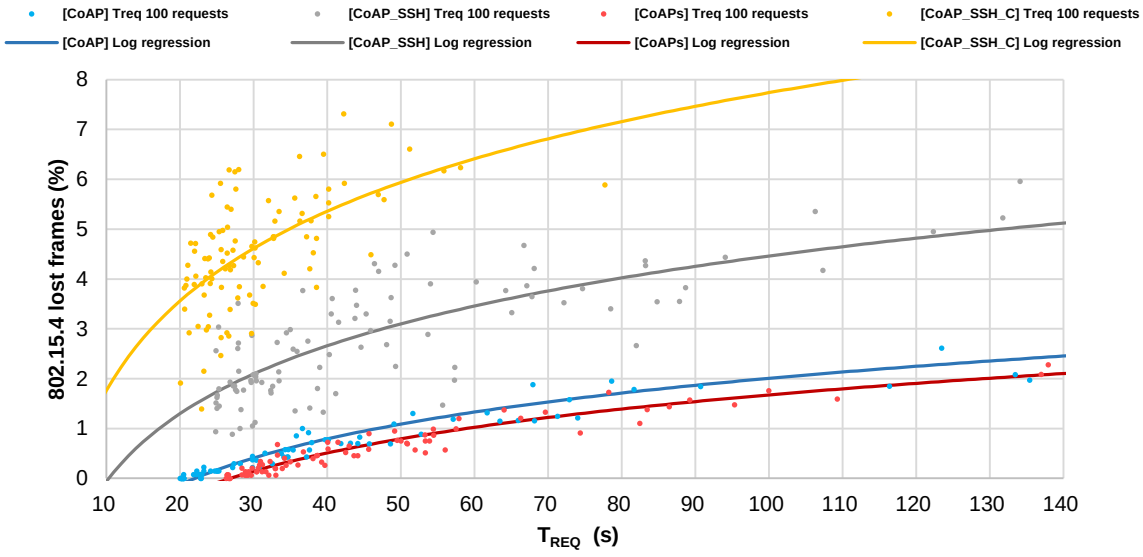


Fig. 4.3-5: Dispersión de T_{req} en los conjuntos de 100 peticiones CoAP GET de 1024 bytes de payload cada una con un LQI promedio causante de la pérdida de ~1% de tramas 802.15.4 de tráfico UDP.

[116] Z. Shelby, K. Hartke and C. Bormann, "The Constrained Application Protocol (CoAP): Messaging Model", 2014, https://tools.ietf.org/html/rfc7252#section-2.1

De estos gráficos puede inferirse el grado de robustez de las comunicaciones frente a la pérdida de paquetes. La robustez se puede vincular unívocamente con la métrica de tramas perdidas en cada *caso*, en vez de tener que utilizar el LQI como métrica de pérdidas, cuando la causa principal de la variabilidad de T_{req} se deba exclusivamente a las pérdidas de tramas del canal por variación de su capacidad.

En la configuración de este escenario, en la que únicamente existe un emisor y un receptor (ambos estáticos), esta condición puede validarse si existe una alta correlación entre las tramas perdidas y el T_{req} observado. Este hecho se confirma afirmativamente gracias al F-test [163], pero sólo en los *casos* en los que se utilizó el mecanismo más conservador de control de congestión, el de CoAP y CoAPs. Tras aplicar el F-test en dichos casos se concluye que sus resultados no se encuentran distribuidos aleatoriamente con una certeza de entre el 90% y el 95%. Por esto último resulta factible asumir que, en este protocolo de pruebas particular, para un valor específico de calidad del canal LQI (que se desconoce) existe un valor bastante concreto del porcentaje de tramas perdidas 802.15.4 para los casos CoAP y CoAPs respectivamente. Debido a la relación entre T_{req} y la métrica de tramas perdidas, se puede afirmar de igual modo que, para un porcentaje específico de tramas perdidas 802.15.4, existe también un valor bastante concreto de la capacidad del canal. En nuestro caso, este hecho se puede apreciar claramente en la Fig. 4.3-5, pero también en la Fig. 4.3-3, en donde queda patente la afinidad que tienen CoAP y CoAPs con sus respectivas funciones de regresión.

El canal inalámbrico puede considerarse como un proceso estocástico de propiedades desconocidas y presumiblemente no estacionario, ni si quiera en sentido amplio. Por ese motivo, no resulta factible calcular las propiedades estadísticas características del canal, tal como su capacidad (capacidad ergódica), ni si quiera tras haber promediado suficientes casos de desvanecimientos (*fading episodes*) [164].

No obstante, si las características estadísticas del canal, aun siendo dependientes del tiempo progresan con suficiente lentitud, se puede llegar a considerar dicho canal como un proceso estacionario en sentido local (*local sense stationary process*) [165]. Por consiguiente, es posible diseñar un escenario de pruebas en el que los cuatro *casos* se alternen, de tal manera que cada caso experimente interferencias similares estadísticamente (*fading*). Para garantizar este efecto, la evolución del canal deberá ser lo suficientemente lenta como para permitir que las interferencias afecten de forma prácticamente equivalente de forma estadística a largo plazo a cada uno de los cuatro *casos* durante las múltiples iteraciones (100 en el test presentado).

Este comportamiento se puede observar en la Fig. 4.3-6, en la que se muestra el comportamiento análogo del promedio acumulado de pérdidas de trama de los *casos* CoAP y CoAPs. Esto explicaría por qué los porcentajes de tramas perdidas 802.15.4 (porcentaje inherente a la capacidad y calidad del canal) es casi el mismo en los *casos* CoAP y CoAPs tras 100

iteraciones. Este resultado justifica la similitud observada en ambos *casos* entre las pérdidas de tramas 802.15.4 (inherentes a la calidad y capacidad del canal) tras 100 iteraciones:

- [CoAP *lost frames avg*.]: 1.014%
- [CoAPs *lost frames avg*.]: 0.986%

Además, como los cuatro *casos* se encuentran alternados en las series de test Fig. 4.3-1, resulta razonable asumir que tanto CoAP_SSH como CoAP_SSH_C también fueron igualmente afectados por el canal durante la totalidad de los test. Por consiguiente, es factible comparar estadísticamente el comportamiento de los cuatro *casos* distintos en diferentes valores de LQI, al considerar que la métrica "porcentaje de pérdida de tramas 802.15.4" puede considerarse como una referencia indirecta del LQI del canal o de su capacidad.

Por otro lado, los dos *casos* en los que se utiliza SSH, no presentan una fuerte correlación y, por tanto, exiten diversos test con valores similares de T_{req} que experimentan un amplio rango de valores de porcentaje de error: existe en efecto una dispersión. Esto indica que esos dos *casos* (CoAP_SSH y CoAP_SSH_C) son más resistentes a interferencias que CoAP o CoAPs: incluso aunque el LQI disminuya (y por tanto aumente el porcentaje de tramas perdidas para CoAP y CoAPs), los *casos* basados en SSH son capaces de adaptarse mejor a dichas interferencias y hacer un uso más eficiente del canal gracias al algoritmo por defecto de control de congestión de TCP reno y su capacidad de adaptarse mejor a los cambios del canal.

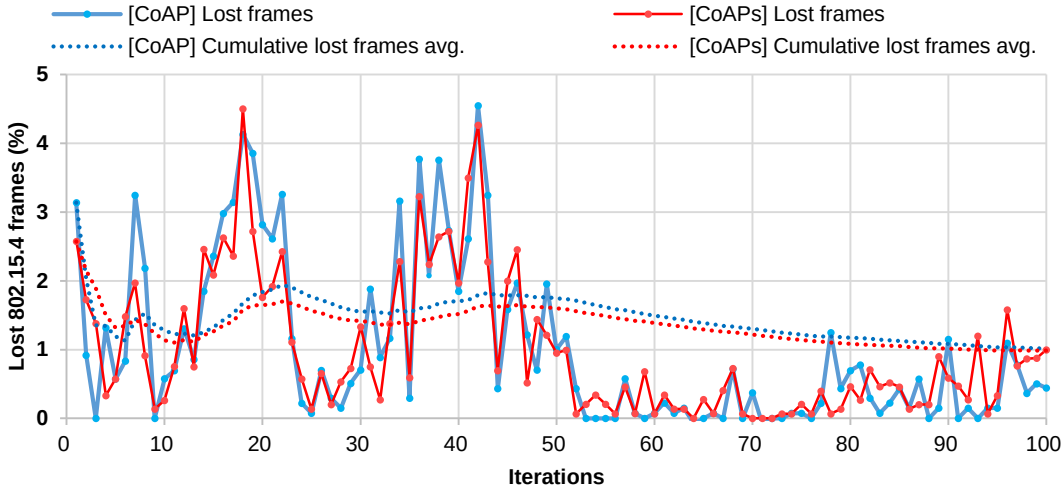


Fig. 4.3-6: Porcentaje promedio acumulado de pérdida de tramas 802.15.4 según la progresión de las 100 iteraciones de test para cada caso.

4.3.1.3. Análisis de los efectos de las interferencias

Los resultados presentados en la Fig. 4.3-5 ponen de manifiesto la incapacidad de CoAPs para mantener los valores de T_{req} si aumentan los efectos de las interferencias. Por otro lado, las nubes de puntos verticales en la parte izquierda de la gráfica, relativas a los *casos* con SSH, no solo presentan una mejor capacidad de aprovechamiento de la capacidad del canal, sino también una mejor resistencia frente a las interferencias: para unos valores similares de T_{req}, existen

múltiples valores de porcentajes de tramas perdidas. En la Fig. 4.3-7 se muestra la capacidad de CoAP_SSH_C de rebajar drásticamente el *overhead* y *payload* por medio de la compresión. Esta característica de SSH prueba ser fundamental en este caso: se produce una reducción del volumen de información por enviar que, a su vez, reduce las probabilidades de interferencia y el tiempo requerido para completar cada petición.

La compresión también puede contribuir a reducir la carga de trabajo de distintos componentes, tanto *software* como *hardware*, por periodos de tiempo más largos. La energía extra que puedan requerir los módulos *hardware* o *software* se compensa por la energía ahorrada en las interfaces de red: se requieren menos transmisiones y escuchar durante menos tiempo [166] [107] (Sección 4.2.2).

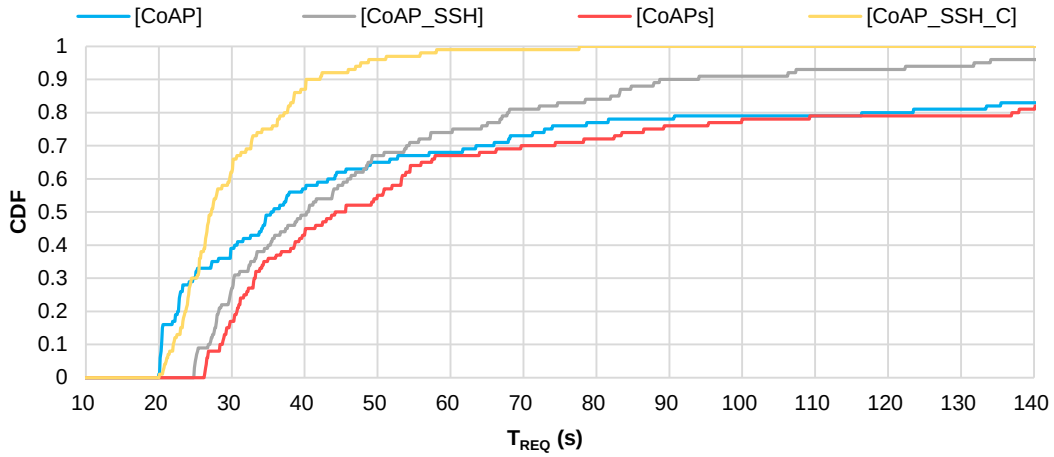


Fig. 4.3-7: CDF de T_{req} de 100 peticiones GET de 1024 bytes de payload cada una con un LQI promedio, causante de la pérdida de ~1% de tramas 802.15.1 relativas al tráfico UDP.

En la Tabla IV se presenta un resumen del promedio de los resultados obtenidos. En este escenario, el *overhead* es superior: aumenta el número de retransmisiones solicitadas debido particularmente a la pérdida de fiabilidad del canal. Sorprendentemente, CoAP_SSH mejora de forma significativa los tiempos de transmisión requeridos cuando se comparan con los de CoAPs (hasta un 36%). Si las funcionalidades de compresión se activan, esta cifra aumenta hasta el 63%.

TABLA IV
RESUMEN DE LOS RESULTADOS DEL 2º ESCENARIO

Caso	Error de trama promedio (%)	T_{req} promedio (s)	ΔP* (%)	*Overhead* promedio (%)
CoAP	1,02	76	8,42	25,72
CoAPs	0,99	83	0	34,81
CoAP_SSH	2,91	53	36,30	89,71
CoAP_SSH_C	4,55	30	63,33	-4,54

* $\Delta P\ (\%) = 100*(T_{req}\_CoAPs - T_{req})/T_{req}\_CoAPs$

4.3.2. **Análisis de los resultados**

CoAP, es un protocolo de nivel de aplicación frecuentemente usado en dispositivos con capacidades limitadas como sustituto de HTTP. Este protocolo proporciona soporte a servicios IoT de diversas categorías, y el desacoplo de seguridad demuestra proporcionar ventajas fundamentales que propician la elaboración de *framework* de certificación nuevos que permiten abordar la problemática regulatoria de la seguridad y privacidad de una forma generalista pero funcional.

Los estudios presentados en este capítulo proporcionan pruebas de la factibilidad del uso del desacoplo de la seguridad en entornos que utilicen CoAP. Estos resultados han permitido caracterizar las ventajas que este tipo de esquemas pueden ofrecer a los dispositivos IoT que utilicen 802.15.4 en presencia de interferencias por comunicaciones 802.11. En particular, se ha evaluado el rendimiento de la implementación generalista basada en IoTsafe, frente a un diseño tradicional basado en comunicaciones seguras por DTLS. Los resultados muestran que este tipo de implementaciones (pese a estar sin optimizarse) son totalmente factibles, permitiendo a las aplicaciones del dispositivo IoT y al *software* del servidor comunicarse de forma transparente y por medio de protocolos inseguros (sin DTLS), confiando todos los aspectos relativos a la autenticación y seguridad a IoTsafe sin requerir la inclusión de ninguna otra prestación en su diseño. Se ha visto que este esquema no penaliza el rendimiento de las comunicaciones en entornos con pérdidas, pese a incrementarse el *overhead* hasta en un 39%, si se compara con la implementación estándar basada en CoAPs.

En escenarios con pérdidas debidas a las interferencias derivadas de las comunicaciones 802.11, el esquema de comunicación propuesto basado en el desacoplo de seguridad permite reducir el tiempo de transferencia en un 36% con respecto a CoAPs, incluso cuando el *overhead* de CoAP_SSH frente al de CoAPs se incrementa en un 54%. Si las prestaciones de compresión de SSH se activan, el *overhead* soportado es el menor de todos los *casos* (incluso hasta negativo, gracias a la alta capacidad de compresión del *payload*) y el rendimiento en términos de tiempo de transferencia aumenta hasta en un 14% en entornos sin pérdidas y hasta un 63% cuando el canal experimenta episodios de *fading* con una pérdida asociada de ~1% de las tramas 802.15.4 de las comunicaciones de los *casos* CoAP o CoAPs.

Estos resultados ponen de manifiesto la importancia del algoritmo de control de congestión y de cómo una solución tecnológica basada en el desacoplo de la seguridad, que puede contribuir a mejorar la tecnología IoT sin necesidad de modificar el diseño de las aplicaciones o protocolos existentes.

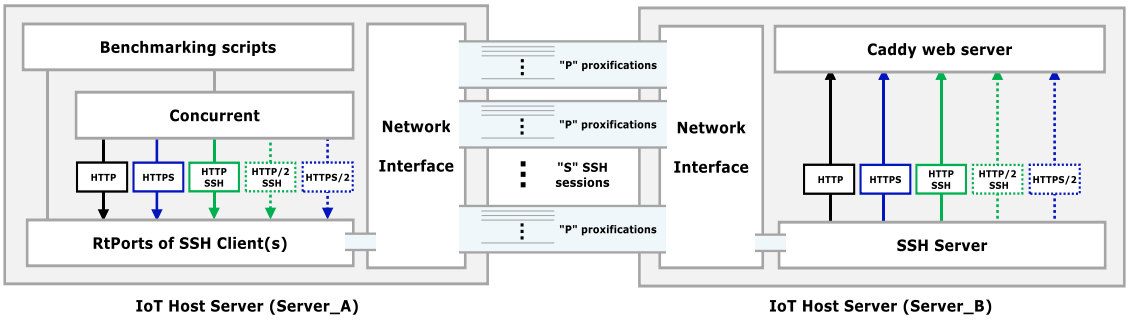


Fig. 4.4-1: Diagrama del escenario de pruebas utilizado para llevar a cabo la comparación de desempeño entre SSH y TLS en comunicaciones basadas en HTTP en el Servidor.

4.4. ESTUDIO DEL RENDIMIENTO DEL SERVIDOR IOT EMPLEANDO UN ESQUEMA DE COMUNICACIÓN BASADO EN PROXIFICACIÓN SSH EN SUSTITUCIÓN DE TLS

Hasta ahora hemos estudiado el rendimiento de IoTsafe en lo que se refiere a la transmisión de datos por el canal inalámbrico. Para completar este estudio presentaremos ahora otro escenario, que se define para caracterizar mejor las limitaciones de SSH en servidores IoT y la penalización potencial en rendimiento y recursos consumidos que el sistema debe soportar en relación a las proxificaciones establecidas, las proxificaciones activas y el tipo de implementación SSH usada.

El escenario de pruebas propuesto se describe en la Fig. 4.4-1 y comprende dos servidores IoT capaces de procesar múltiples comunicaciones en paralelo. Estos servidores son en realidad máquinas virtuales basadas en el sistema *Kernel Virtual Machine* (KVM)[117] proporcionado por Linode como servicio en la nube[118]. Cada máquina virtual consiste en un Linode 16GB, que comprende un sistema operativo Centos 7, 16 GB de memoria RAM, 6 núcleos virtuales Intel(R) Xeon(R) CPU E5-2680 v3 2.50 GHz (un hilo, un núcleo), y una conexión local media de 1 Gbps entre ellos. El primer servidor (*Server_A*) se utiliza para lanzar un número determinado de clientes SSH, y el segundo (SSH_B) genera múltiples proxificaciones y sesiones SSH como resultado de las comunicaciones entrantes desde *Server_A.*

Si el *Server_B* requiere trasladar las proxificaciones de las que se encarga hacia otro servidor distinto, éste jugaría el papel de *Server_A,* generando todas las instancias requeridas de clientes SSH para enviar esas proxificaciones. De ese modo, el escenario de pruebas propuesto sirve para caracterizar el peor caso de los dos posibles: un servidor recibiendo múltiples proxificaciones IoT, y un servidor enviando múltiples proxificaciones hacia otro servidor. Este último caso se puede observar en infraestructuras basadas en IoTsafe, en las que un servidor IoT completo debe de ser

[117] "Kernel-based Virtual Machine", https://www.linux-kvm.org/page/Main_Page
[118] "Linode", https://www.linode.com/

capaz de ceder proxificaciones locales a servidores remotos de forma segura y efectiva (Anexo C, ejemplos 6, 7, 8 y 9). Otras variables que también se toman en consideración son:

- El número de sesiones establecidas.
- El número de proxificaciones establecidas (en una única sesión o por sesión).
- El número de proxificaciones activas (en una única sesión o por sesión).

Esta diferenciación permite evaluar el rendimiento de la solución propuesta y sugerir mejoras y optimizaciones que permitan garantizar una solución más completa y escalable.

4.4.1. **Análisis de las posibles limitaciones de escalabilidad existentes en los gateways y servidores empleando comunicaciones proxificadas**

En esta sección nos planteamos responder a esta cuestión:

C9: ¿Qué desafíos y limitaciones afrontaría la aplicación de este esquema de seguridad en los *Gateways*/servidores?

El principal objetivo de los test presentados en esta sección persigue el caracterizar el rendimiento de la arquitectura propuesta en el lado del servidor, para determinar su escalabilidad. Por ello, se estudian las limitaciones potenciales derivadas del uso de SSH para comunicaciones basadas en *port-forwarding*, se analiza la dependencia de dichas limitaciones en función del número de sesiones, proxificaciones y comunicaciones activas, y también el impacto según el tipo de implementación usada (OpenSSH o Dropbear).

El primer parámetro a caracterizar es el consumo de memoria RAM. Además de esto, se monitoriza el tiempo de ejecución en distintos test, para determinar mejor la dependencia entre este parámetro y la implementación particular de SSH elegida, el tamaño del *payload* de cada petición y la distribución de las proxificaciones en sesiones paralelas. Estos resultados permitirán detectar otras posibles limitaciones de SSH, como la relación que tiene su rendimiento con el número de comunicaciones simultáneas activas y su distribución en canales y sesiones SSH (una sesión SSH se establece entre el cliente SSH y el servidor SSH y puede albergar múltiples canales de comunicación, cada uno de ellos necesario para establecer una proxificación).

Estas son las características de los dos servidores empleados:

- *Server_A,* que utiliza Centos 7 y alberga ambas implementaciones de servidores SSH. La primera es DropBear 2018.76, compilada en Ubuntu 14.04.5 LTS como *stand-alone* (siendo una versión portable y también con las prestaciones de *debug* activadas). Esta compilación se realiza utilizando una versión modificada de la librería *eglibc* v3.10.0 en la que la variable FD_SETSIZE se ha incrementado para sortear la antigua limitación de 1024 descriptores de ficheros (y *socket*) relativa a la función *select()* de POSIX. El código fuente de DropBear también se modifica para evitar esta limitación, junto con otras variables como MAX_LISTENERS, MAX_CMD_LEN and MAX_CHANNELS para

que permitan llevar a cabo los test planteados. La segunda implementación utilizada es OpenSSH_7.4p1 con OpenSSL 1.0.2k-fips incluida en Centos7 por defecto.

- *Server_B,* también utiliza Centos 7. Este servidor se utiliza como generador de tráfico SSH para ambos tipos de implementación SSH. El cliente DropBear se ha compilado utilizando las mismas características y funcionalidades que se definieron para el servidor DropBear en el *Server_A.*

4.4.2. **Caracterización del consumo de memoria RAM**

El primer aspecto analizado concierne a la memoria RAM requerida por el funcionamiento de los servidores SSH, así como su relación con el número de proxificaciones activas.

Para proceder con este análisis, cada cliente SSH en el *Server_A* simula el establecimiento de una sesión de un dispositivo IoT hacia el *Server_B,* de modo que el número de sesiones entre los servidores represente el número de dispositivos IoT conectados al *Server_B*. Cada sesión se utiliza para habilitar un único canal de comunicación para establecer una proxificación directa (un Dispositivo IoT, una Sesión con el *Server_B*, una única proxificación). Este es el procedimiento ordinario por el que un dispositivo IoT, utilizando la arquitectura propuesta, se conectaría a un servicio del *Server_B* (plataforma IoT) de forma segura por medio de una conexión local entre la aplicación IoT del dispositivo y el cliente SSH dentro del mismo dispositivo con la proxificación abierta.

Este test persigue por tanto evaluar el consumo de RAM atribuible a cada una de las proxificaciones derivadas de cada dispositivo IoT y que debe manejar el servidor IoT (en el caso en que cada dispositivo IoT únicamente requiera para funcionar una única proxificación directa). Las proxificaciones, por tanto, se distribuyen en sesiones SSH del siguiente modo:

- **S**: Número total de sesiones SSH que maneja el servidor IoT.
- **P**: Número total de proxificaciones por sesión SSH.
- **T**: Número total de proxificaciones en el test.
- **N**: Número de proxificaciones inactivas

En este caso, P=1 y S es igual al número total de proxificaciones establecidas con el *Server_B* (equivalente al número total de dispositivos IoT), todo según la Fig. 4.4-1. El número de proxificaciones establecidas se aumenta progresivamente para estudiar el crecimiento en el consumo de RAM.

El test también permite una comparación entre OpenSSH y Dropbear. En todas las medidas, bien para OpenSSH o DropBear, tanto el cliente como el servidor son del mismo tipo. De ese modo, si la solicitud la realiza OpenSSH, el cliente también es OpenSSH. Si la solicitud se realiza desde DropBear, el servidor también será de tipo DropBear.

Los servidores IoT también pueden establecer proxificaciones entre ellos, particularmente cuando un pequeño servidor IoT actúa como *gateway*. Este caso particular se pone a prueba también estableciendo canales de comunicación agregados en una misma sesión SSH (en la gráfica, denominado como "*aggregated*"). En este caso particular, se establece una única sesión del *Server_B* al *Server_A* que incluye múltiples proxificaciones y, por tanto, S=1 y P es igual al número de proxificaciones establecidas; todo de nuevo según la Fig. 4.4-1.

Los resultados de este primer test se presentan en la Fig. 4.4-2, que muestra el consumo de RAM por proxificación en cada caso. Una vez establecidas todas las proxificaciones, cada una de ellas es puesta a prueba por medio de una petición HTTP GET de 1 kbyte de *payload*. Estas solicitudes se realizan a través de cada una de las proxificaciones y en paralelo, usando un único programa como cliente escrito en Golang y denominado "*Concurrent*" en el *Server_B.* El *Server_A* alberga el servidor web Caddy, al cual se accede a través de cada proxificación (Caddy está también escrito en Golang).

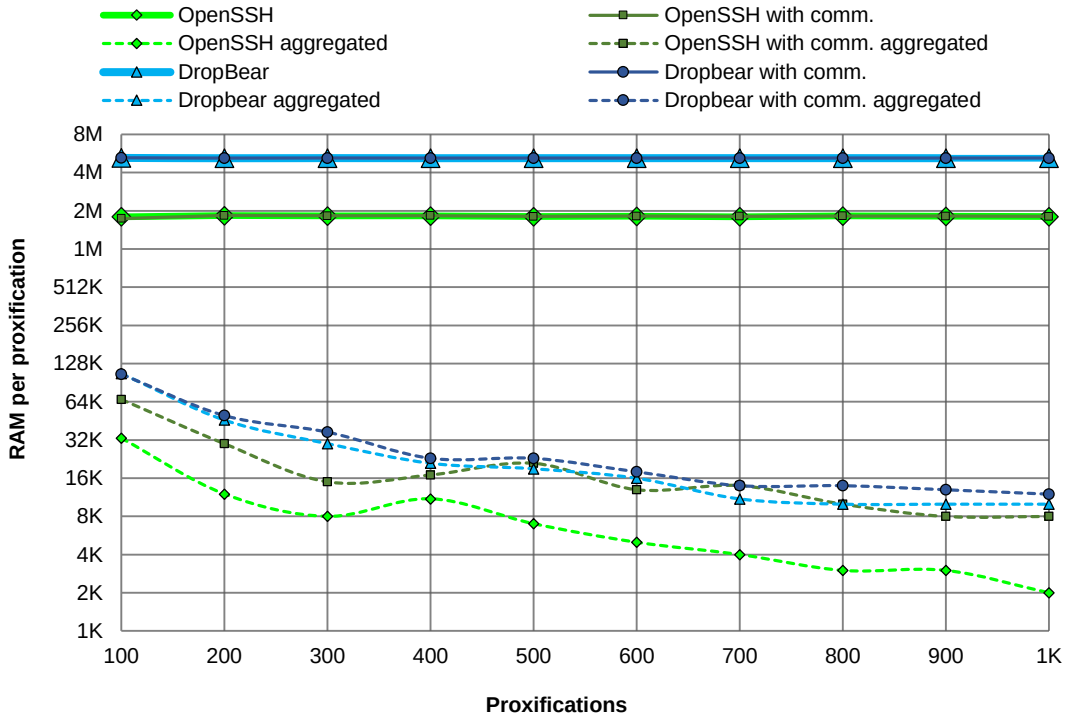


Fig. 4.4-2: Comparativa del consumo de RAM requerido en cada caso por número de proxificaciones y configuración elegida. Cada proxificación procesa en paralelo una petición HTTP GET que devuelve una respuesta de 1Kbyte de payload.

La medición de consumo de RAM se realiza de forma global (en todo el sistema operativo) mediante el comando *free*[119]. El *Server_A* se utiliza de forma dedicada exclusivamente para este *benchmark* y el consumo de RAM global es monitorizado continuamente durante todo el proceso. Su valor máximo se obtiene en tres partes diferentes de cada test:

- C_{base}: Consumo base obtenido antes de comenzar el test (y después de haber realizado "sync[120]").
- C_{max}: Durante el *benchmark* se mide periódicamente el consumo de RAM. C_{max} representa el valor máximo registrado durante el proceso.

El consumo de RAM en cada caso se determina restando C_{max} - C_{base} y se presenta en la Fig. 4.4-2. El consumo de RAM por proxificación, tanto en Dropbear como en OpenSSH, es extremadamente alto cuando una sesión maneja una única proxificación (P=1, S=N). Este resultado se debe principalmente al hecho de que cada conexión entrante SSH se maneja por el servidor SSH en el *Server_A* como una instancia separada e independiente del servidor SSH principal, para así mantener la separación de privilegios: se instancia un servidor SSH en la cuenta de usuario relativa al usuario autenticado por SSH (Anexo C, ejemplo 1, Fig. C-1). El procedimiento seguido consume una gran cantidad de memoria RAM, algo que resulta notable sobre todo con DropBear, ya que se compiló con las características de *debug* y en modo *stand-alone*, lo que hace que sus requisitos de memoria RAM dupliquen a los de OpenSSH.

El caso en que existiese un único dispositivo IoT y éste realizase todas las proxificaciones hacia el servidor (P=N, S=1), se obtendría el resultado reflejado en las series de datos que denominadas "*comm. aggregated*". En estas circunstancias, el proceso instanciado en el servidor es único y, por tanto, el consumo de RAM aumenta de forma mucho más moderada con el número de proxificaciones, al no existir nuevos *fork* (nuevas instancias) del servidor SSH.

4.4.3. Solución del problema de la falta de escalabilidad por el elevado consumo de RAM

En esta sección se trata de ofrecer una solución pragmática a las limitaciones encontradas en el esquema de comunicación propuesto por IoTsafe, que utiliza implementaciones actuales de SSH. Esta cuestión hace referencia a la C10 descrita en la sección 1.3:

C10: ¿Qué propuestas técnicas se pueden presentar para hacer frente a estas limitaciones?

A pesar de que los resultados en consumo de RAM puedan mejorarse notablemente efectuando optimizaciones importantes en las implementaciones de SSH, el resultado muy difícilmente alcanzaría cifras similares a las que presenta TLS en situaciones similares, con

[119] "free - Display amount of free and used memory in the system", *Linux man page*, 2018, http://man7.org/linux/man-pages/man1/free.1.html
[120] "sync - Synchronize cached writes to persistent storage", *Linux man page*, 2019, http://man7.org/linux/man-pages/man1/sync.1.html

consumos de decenas de Kbytes por sesión. Esto es principalmente causado por el procedimiento de instanciación de cada sesión entrante al servidor SSH. En este proceso, por cada sesión nueva en el *host* IoT, el servidor SSH del *host* hace un *fork* de sí mismo bajo la cuenta de usuario correspondiente a las credenciales usadas en el proceso de autenticación. Esto deriva en un consumo de RAM muy elevado por cada dispositivo IoT, tal y como quedó patente en la Fig. 4.4-2: por cada dispositivo IoT conectado al servidor se requieren entre 2 y 5 Mbytes de RAM.

Para mitigar este efecto, el esquema propuesto puede beneficiarse del mejor comportamiento de las comunicaciones agregadas puesto de manifiesto en esa misma figura. Este esquema resulta trivial en la mayoría de implementaciones actuales de infraestructuras IoT, gracias a que suelen contemplar un elemento intermedio entre los dispositivos IoT y el Servidor: el *Gateway*.

Un *Gateway* de IoT podría actuar como un servidor IoT pequeño, que sirva para concentrar un pequeño número de sesiones SSH, cada una de un dispositivo IoT distinto, y multiplexar las proxificaciones de forma conjunta para mandarlas en una única sesión al servidor IoT. Este esquema de comunicación puede llevarse fácilmente a la práctica aprovechando las características de SSH para agregar múltiples canales de comunicación (múltiples proxificaciones) en una única sesión (comunicaciones agregadas). En este caso, cada proxificación entre el *gateway* y el servidor IoT requiere de mucha menor cantidad de RAM, permitiendo que este esquema de comunicación en dos niveles pueda reducir notablemente los requisitos de RAM.

Para continuar con el análisis del comportamiento de las comunicaciones proxificadas en el servidor, se presenta otro test diferente, que persigue analizar la dependencia del consumo de RAM por cada una de las proxificaciones establecidas según el tamaño del *payload* de la petición cursada por cada proxificación. Para ello, se establecen un total de 100 proxificaciones activas (manejando cada una de ellas una petición HTTP GET) y se procesan de dos maneras diferentes:

- Cada proxificación en una sesión SSH independiente.
- Todas las proxificaciones son cursadas a través de una misma sesión SSH.

El test incluye pruebas con diferentes tamaños de *payload,* para determinar también su posible dependencia con el consumo de memoria RAM.

La Fig. 4.4-3 muestra los resultados de estos test. El consumo de RAM presenta poca dependencia con el tamaño del *payload* cuando se efectúan comunicaciones desde los dispositivos IoT hacia el *Gateway*. Sin embargo, una vez se agregan todas estas proxificaciones en una única sesión SSH, queda patente que OpenSSH se desenvuelve peor que DropBear. Esto es debido principalmente a la manera que tiene OpenSSH de manejar los *buffer* de las ventanas de comunicación de cada canal para cada proxificación. Este problema explica la dependencia lineal del consumo de RAM con el tamaño creciente del *payload* de las peticiones HTTP.

Se propone a continuación otro test diferente para este segundo escenario, con el que poder estudiar mejor la posible penalización en rendimiento (tiempo requerido para la transmisión). Este tiempo es el que una instancia SSH tiene que afrontar cuando coexiste con proxificaciones inactivas

y la posible dependencia de dicho consumo de RAM con el tamaño del *payload*. La motivación de este test radica en la necesidad de caracterizar el coste computacional del manejo de las múltiples proxificaciones abiertas por el sistema, y determinar el impacto que tienen a nivel de *kernel*. Por ello, sólo se establecen proxificaciones agregadas en una única sesión SSH. La sesión SSH tiene 1000 proxificaciones activas (P=1000, S=1) por las que se cursan las peticiones HTTP GET, y en el resto de proxificaciones existentes no se efectuará ningún tipo de petición. Los *payload* analizados serán de 64, 128 y 256 Kbytes.

El tiempo requerido para completar cada test se mide por el programa en GoLang "*Concurrent*" realizado para este benchmark. Este programa es un cliente HTTP que inicia simultáneamente todas las peticiones previstas y monitoriza tanto el *payload* de las respuestas como el tiempo total requerido para completar todas las peticiones HTTP en paralelo. "*Concurrent*" se lanza una vez que todas las sesiones SSH se han establecido. Se realizan un total de 25 test por cada configuración de proxificaciones establecidas. Los resultados de los test son promediados y se representan en la Fig. 4.4-3.

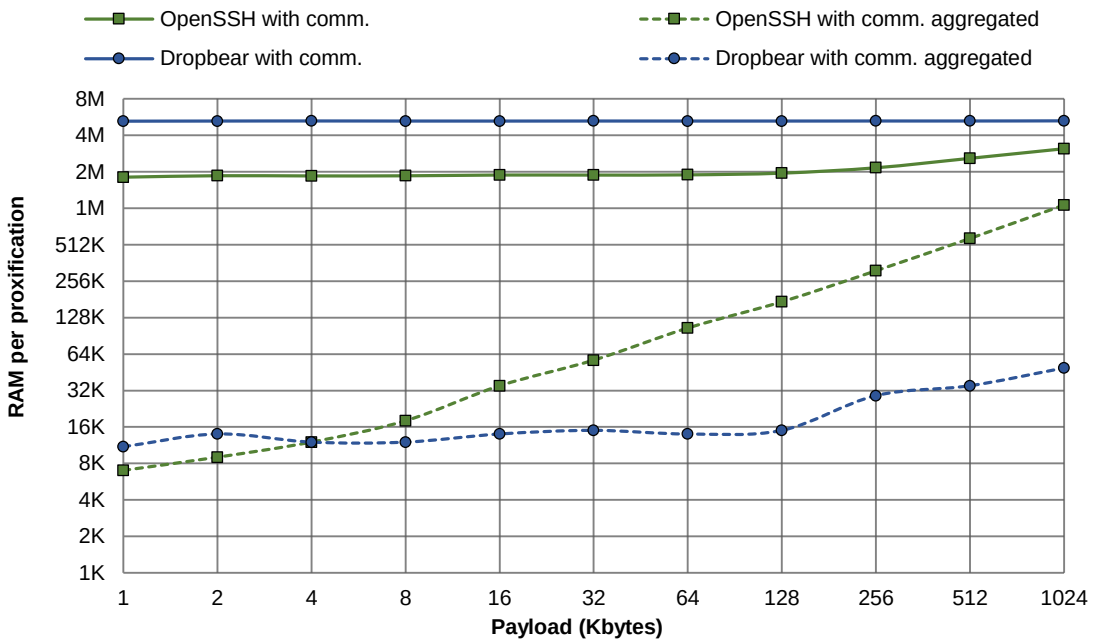


Fig. 4.4-3: Comparativa del consumo de RAM por proxificación en comunicaciones según aumenta el payload de las peticiones HTTP cursadas por la proxificación. Se efectúan 1000 proxificaciones en cada caso y se comparan las dos implementaciones de SSH con dos configuraciones distintas: Una sesión por proxificación (P=1, S=1000) y una única sesión para todas las proxificaciones, es decir, comm. aggregated (P=1000, S=1).

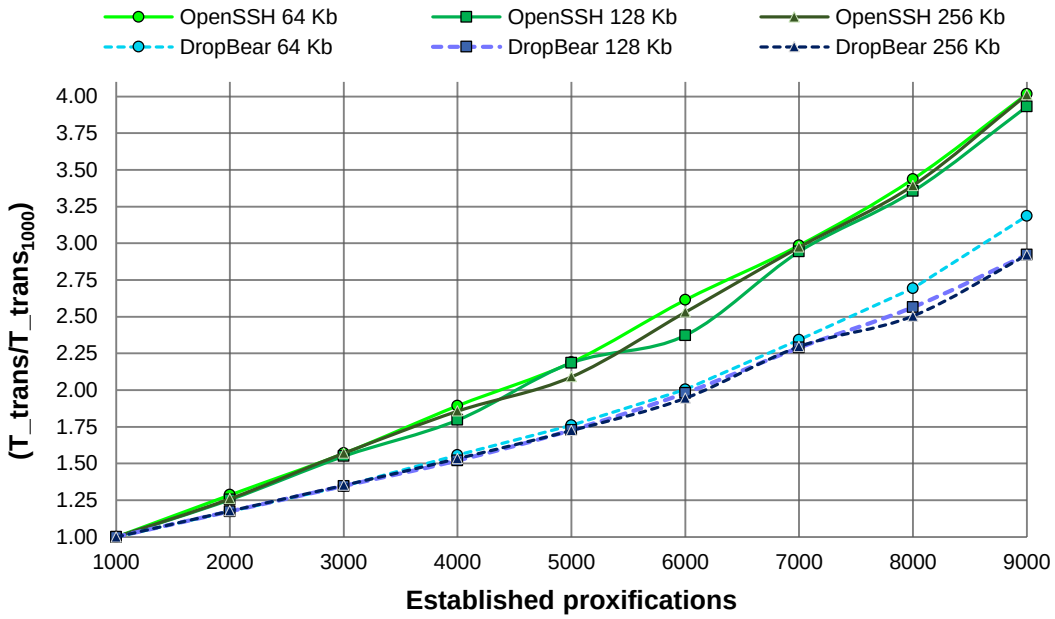


Fig. 4.4-4: Rendimiento normalizado de 1000 proxificaciones activas en función del número de proxificaciones totales establecidas en una misma sesión y la implementación SSH utilizada.

Los resultados muestran una penalización en el rendimiento. Esta penalización se observa al comparar el resultado de dividir todos los resultados promediados por el tiempo requerido para terminar los 25 test del caso en el que sólo hay 1000 conexiones activas y ninguna inactiva. El tiempo que necesita cada configuración en realizar los 25 test se define como *T_trans,* mientras que el tiempo requerido por la configuración particular en la que no hay ninguna conexión inactiva es T_trans_{1000}. Es importante aclarar que cada caso (cada una de las seis series de medidas distinta) tiene su propio T_trans_{1000}.

Estos resultados muestran una escasa dependencia con el tamaño de *payload*, pero por el contrario sí parece que la existencia de proxificaciones inactivas en la misma sesión SSH les afecta. Esta dependencia exponencial está relacionada con la manera en la que SSH maneja cada canal de comunicación. Pese a ser canales de comunicación inactivos, aparentemente ejercen una fuerte penalización sobre el rendimiento global del test, aunque no haya que efectuar ninguna tarea adicional. Es conveniente recordar que los tiempos monitorizados recogen exclusivamente la duración de los test de pruebas (peticiones HTTP GET). El tiempo necesario en establecer las sesiones SSH y las proxificaciones no se contempla.

La causa de la penalización en rendimiento causada por la existencia de canales inactivos se debe a la forma en la que se maneja la apertura y mantenimiento de los *socket* de las proxificaciones. Las implementaciones de SSH están empleando un *framework* I/O desfasado que no funciona bien en situaciones en las que un proceso debe manejar cantidades elevadas de descriptores (*socket*).

La TABLA V se ha elaborado a partir de un escenario de pruebas que comprende 100.000 operaciones aleatorias de monitorización de descriptores contiguos [65]. La tabla presenta la degradación en rendimiento sufrida por monitorizar N descriptores utilizando *select(), poll() y epoll()* según se aumenta el número de descriptores, tomando N=10 como referencia.

TABLA V
DEGRADACIÓN DEL RENDIMIENTO DE LOS DIFERENTES FRAMEWORKS I/O AL GESTIONAR 100,000 OPERACIONES ENTRE N DESCRIPTORES

Número de descriptores a monitorizar (N)	*select*()	*poll*()	*epoll*()
10	1,00	1,00	1,00
100	4,75	4,11	1,02
1.000	57,38	47,95	1,29
10.000	1.622,95	1.273,97	1,61

Tanto *select()* como *poll()* son llamadas al sistema pertenecientes a un *framework* I/O basado en multiplexión, cuya penalización en rendimiento varía exponencialmente en función del número de descriptores a monitorizar (N). Esto se debe principalmente a que "*en cada llamada a select() o poll(), el kernel debe revisar todos los* descriptores *especificados* (manejados por el proceso) *para averiguar si* (los descriptores requeridos) *están listos*" [65]. El *kernel* de Linux, bajo el funcionamiento de este I/O *framework*, debe revisar siempre todos los descriptores que el proceso tiene abiertos, independientemente de con cuáles requiere actualmente interactuar.

Por otro lado, la penalización en rendimiento sufrida por *epoll()* es mucho menor debido a que forma parte de una API I/O dirigida por señales del sistema, en la que los procesos que requieren utilizar un descriptor abierto reciben una señal del sistema enviada por el *kernel* una vez que las operaciones de entrada y/o salida son posibles con el descriptor requerido. Además, la pequeña penalización en rendimiento observada puede reducirse aún más dependiendo de las características de *hardware* en las que se lleve a cabo la prueba. En [65], se expone en relación con *epoll()* que "*la ligera reducción en rendimiento conforme N aumenta se debe posiblemente al resultado de alcanzar el límite de caché de CPU en el sistema de pruebas*".

Debido a las limitaciones derivadas de las implementaciones SSH existentes, una sesión SSH no es capaz de manejar grandes cantidades de canales de comunicación para realizar las proxificaciones, ya que éstas funcionan con *socket* que son descriptores que se encuentran regidos por un *framework* I/O obsoleto. Por otro lado, utilizar una sesión SSH por proxificación (un *socket* por proceso) de forma general, no resultaría factible debido al gran consumo de memoria RAM que eso conllevaría. En los siguientes test se evalúa la menor degradación de rendimiento ocurrida cuando se efectúan múltiples sesiones SSH en paralelo, que combinan proxificaciones activas e inactivas para analizar el impacto de distribuir dichas proxificaciones en más sesiones SSH. Para ello, los test se realizarán de la siguiente manera:

- Cada sesión SSH contará con 128 proxificaciones en total, de las que sólo 64 estarán activas. Estos valores son constantes para todos los casos durante estas pruebas. El número total de proxificaciones activas siempre es de 64*128=8.192.
- Para cada *caso* y para cada uno de sus 25 test, se efectúa en paralelo una petición HTTP GET de un *payload* de 64 kbytes por cada una de las 8.192 proxificaciones activas.
- En el test existen dos tipos de proxificaciones, las activas y las inactivas. Una proxificación activa tramita peticiones HTTP GET. Por otro lado, una proxificación inactiva no se utiliza durante el procedimiento. En este test, el número total de proxificaciones activas se mantiene constante, aunque el número total de proxificaciones crezca con cada configuración diferente a base de proxificaciones inactivas. El trabajo total realizado en términos de peticiones HTTP GET es el mismo en cada realización del test, pero el número de proxificaciones establecidas y procesos se va aumentando con consiguiente realización para medir la penalización derivada de gestionar descriptores abiertos en múltiples procesos en ejecución.
- En vez de aumentar el número de proxificaciones en una misma sesión SSH, aumenta el número de sesiones SSH establecidas en paralelo. Las 8.192 proxificaciones activas se distribuyen entre las sesiones.

Debido a las limitaciones de las implementaciones de SSH, una sesión no puede manejar grandes cantidades de *socket*. Por otro lado, usar una sesión por proxificación puede no ser factible en todas las circunstancias, debido a que el consumo de RAM sería muy elevado. Con los test propuestos se evalúa el impacto en la degradación de rendimiento conforme se incrementa el número total de proxificaciones, pero sin aumentar el trabajo total realizado (número de peticiones o de proxificaciones activas totales).

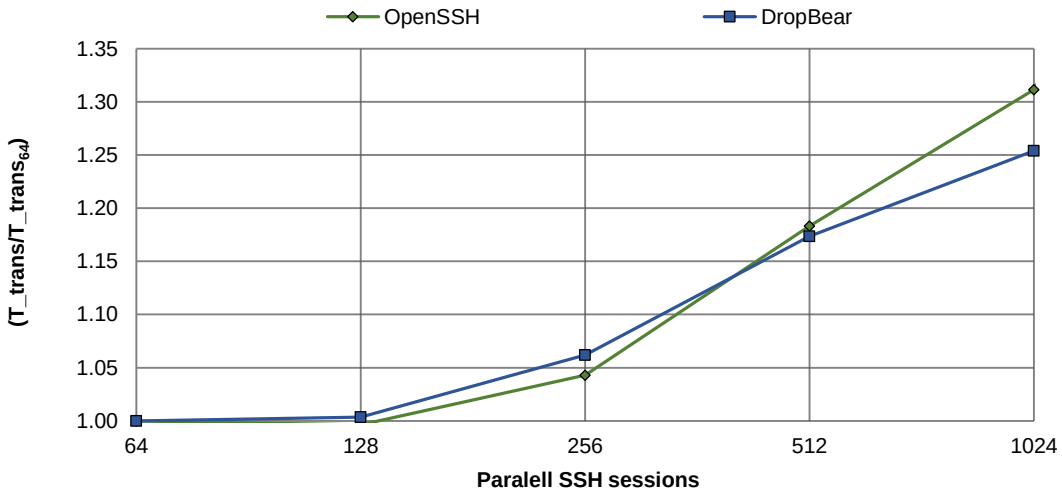


Fig. 4.4-5: Evolución del rendimiento normalizado de 64 sesiones SSH en paralelo (S=64) con 128 proxificaciones activas cada una de ellas en función del aumento del número de sesiones SSH en paralelo con 128 proxificaciones inactivas cada una de ellas. Cada proxificación activa maneja una petición simultanea HTTP de 64 Kbytes de payload.

Tal y como se aprecia en la Fig. 4.4-5, según crece el número de procesos, el *kernel* tendrá que dividir los recursos disponibles de la CPU entre ellos, conduciendo a un escenario de ineficiencia. La penalización de rendimiento se percibe incluso cuando el número de peticiones en paralelo no aumenta. Sin embargo, dicha degradación se produce más lentamente que en el ejemplo en el que todas las proxificaciones se concentraban en una única sesión.

En la Tabla VI se presenta una comparación entre ambas situaciones, para mostrar más claramente el efecto de distribuir las proxificaciones en grupos manejados por sesiones SSH (Fig. 4.4-5) en vez de manejarlas todas únicamente en una sola sesión (Fig. 4.4-4). En esa tabla, N representa el número de proxificaciones sin usar (establecidas – usadas). Cabe destacar, no obstante, que los test de rendimiento presentados en la Fig. 4.4-4 de la serie de 64 Kbytes tuvieron sólo 1.000 proxificaciones activas mientras que los de la Fig. 4.4-5 tuvieron 8.192. La Tabla VI muestra el porcentaje de penalización en rendimiento según aumentan las proxificaciones sin usar.

TABLA VI
PORCENTAJE DE LA PENALIZACIÓN DEL RENDIMIENTO (%) DE DISTINTAS CONFIGURACIONES SSH PAR A N PROXIFICACIONES ESTABLECIDAS SIN USAR

Configuraciones	N=1.000	N=2.000	N=4.000	N=8.000	N=24.576	N=57.344	N=122.880
OpenSSH (64kb) P=N, S=1	29%	57%	119%	302%	-	-	-
DropBear (64kb) P=N, S=1	17%	35%	76%	219%	-	-	-
OpenSSH P=128, S=N/128	~ 0%	~ 0%	~ 0%	~ 0%	4%	18%	32%
DropBear P=128, S=N/128	~ 0%	~ 0%	~ 0%	~ 0%	6%	17%	25%

Es importante notar que solo se produce una degradación de entre el 25-32% cuando N alcanza un valor total de (1.024*128)-8.192=122.880 proxificaciones sin usar distribuidas en un total de 1.024 sesiones (Fig. 4.4-5). Por otro lado, cuando únicamente se utilizaba una sesión SSH se producía una degradación del rendimiento de entre 290% y el 400% con únicamente 9.000-1.000=8.000 proxificaciones inactivas (Fig. 4.4-4). Por consiguiente, queda patente la posibilidad de escalar la arquitectura propuesta de comunicación en los servidores IoT, combinando múltiples proxificaciones en múltiples sesiones SSH.

4.4.4. **Estimación de la relación óptima entre P y S**

Las siguientes pruebas se presentan con el objeto de estimar la relación óptima existente entre estos dos parámetros: Las proxificaciones por sesiones “P” y el número de sesiones “S”. En esta prueba, se manejan un total de 10.000 proxificaciones activas, en distintas configuraciones. En cada configuración se incrementa el total de sesiones SSH (S), en las que hay que distribuir el total de proxificaciones (P) cumpliéndose que P*(S-1) <= 10.000 y P*S >= 10.000. Dado que las diferentes configuraciones posibles no son divisores enteros de

10.000, existirán configuraciones en las que tenga que haber proxificaciones sin usar, que serán como máximo P-1 (el número de proxificaciones por sesión menos una).

El tiempo promedio computado para terminar una petición HTTP de 64 Kbytes a través de cada una de las 10.000 proxificaciones activas se denota como *T_trans*. El eje Y representa el rendimiento de cada test frente al tiempo requerido por la configuración óptima denominada T_trans_{min}. Este valor es definido como el *T_trans* menor de todas las series de test y, por consiguiente, todas las demás configuraciones sufren una cierta penalización en rendimiento frente a esta configuración que obtiene el tiempo mínimo. Tanto la serie de test de OpenSSH como la de DropBear poseen su propio $T\_trans_{min.}$ En la Fig. 4.4-6 se presentan los resultados empleando ambos ejes en escala no lineal. Según esos resultados, existe una relación razonablemente óptima entre el número de proxificaciones por sesión "P" y el número de sesiones "S" que sería P/S=79/128, tanto para OpenSSH como para DropBear.

Una vez se ha determinado una relación óptima entre P y S, se puede proceder con una comparación entre el esquema de comunicación segura tradicional TLS que utilizan las aplicaciones basadas en HTTP, y el esquema de comunicación basado en el desacoplo de seguridad empleando el protocolo SSH, a través de sus implementaciones principales en OpenSSH y DropBear. Para ello se presenta un test en el que se mide el tiempo dedicado por cada esquema, permitiendo computar la métrica denominada Peticiones Por Segundo (*Request Per Second*, RPS). Esta métrica representa la cantidad de peticiones HTTP de 1 Kbyte que el servidor IoT puede procesar en paralelo en un segundo. El objetivo es medir el rendimiento del servidor según el esquema de comunicación elegido entre el servidor IoT y los dispositivos. Por todo ello, se definen ciertos aspectos que deben cumplirse:

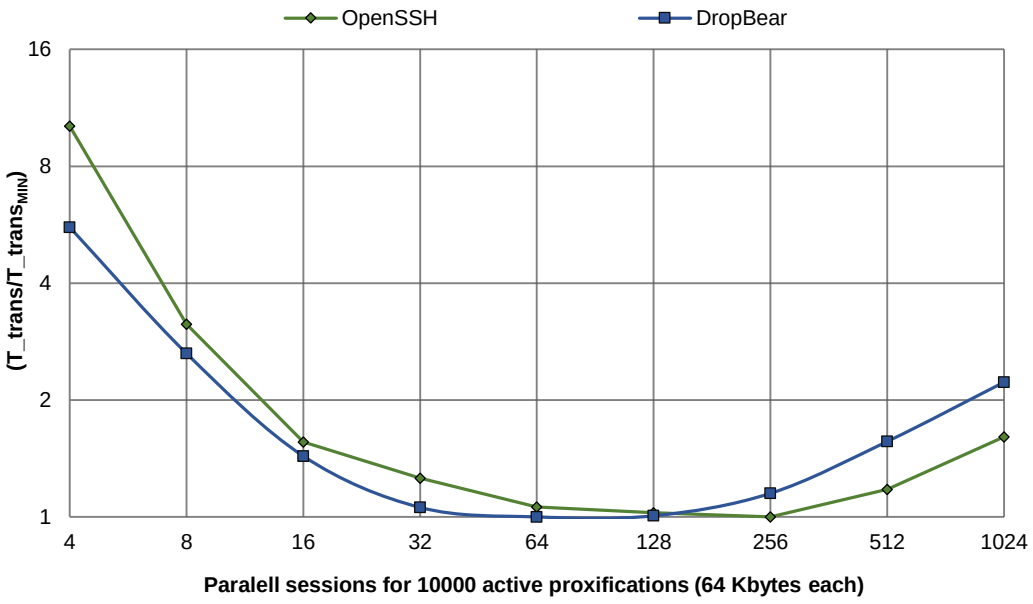


Fig. 4.4-6: Rendimiento normalizado de las implementaciones SSH con diferente configuraciones P y S para conseguir 10.000 proxificaciones activas. Cada proxificación activa maneja una petición simultanea HTTP de 64 Kbytes de payload.

- Cada petición directa, empleando HTTPS de "*Concurrent*" (Fig. 4.4-1) en el *Server_B* hacia el servidor Caddy en el *Server_A,* debe realizarse independientemente. Por consiguiente, HTTPS/2 no podrá beneficiarse de sus características inherentes de multiplexación. Los test se realizan de este modo debido a que pretenden representar peticiones de múltiples dispositivos IoT independientes y que, por tanto, no pueden multiplexarse porque provienen de dispositivos distintos. Las características de multiplexación de HTTPS/2 se ponen en funcionamiento en los casos en los que se solicita una página web con multitud de objetos, imágenes, etc. Tradicionalmente el explorador web debía realizar una petición HTTP por cada objeto, pero HTTP/2 permite unificarlas todas en una sola. Este comportamiento no se aplica en el test presentado y, por consiguiente, las peticiones se realizan de manera que no puedan multiplexarse. Esto se consigue ejecutando Caddy en múltiples direcciones IP locales y puertos, y dirigiendo cada una de las peticiones en paralelo que se estipulen hacia un *socket* diferente del servidor. Esto se lleva a cabo tanto en peticiones directas HTTP, HTTPS y HTTPS/2 como peticiones proxificadas a través de SSH, para suprimir en la comparación la posible penalización de abrir múltiples *socket* de escucha en el servidor.

- Las peticiones HTTPS se realizan sin validar los certificados SSL ante una autoridad certificadora. Esta tarea se prevé que la haga únicamente el cliente (apenas interviene el servidor IoT) y, de realizarse estrictamente, retrasaría cada petición HTTPS también como resultado de esto último. Dado que el objetivo de este test se centra en medir el rendimiento del servidor, el tiempo para la validación de los certificados HTTPS por cada petición no se debe contar, para no perjudicar al caso HTTPS (HTTP usando TLS).

- Las resoluciones DNS inherentes a las peticiones no se realizan, para evitar otra penalización a las solicitudes HTTPS y HTTPS/2 que tampoco sería procedente. Cada resolución DNS (incluso aunque se encuentre en cache) que debería ejecutar el cliente HTTP, incrementaría el tiempo requerido por cada petición y penalizaría la medida de rendimiento del servidor.

- Existen distintos tipos de optimizaciones en el *kernel* que buscan incrementar la seguridad[121], o evitan que se produzcan ataques de denegación de servicio[122] por peticiones masivas sincronizadas contra un recurso ofrecido. Para evitar que dichas limitaciones interfieran en los test de HTTP, HTTPS y HTTPS/2, se desactivan durante la evaluación de los test de este caso.

- Las peticiones HTTP protegidas a través de proxificaciones provistas por OpenSSH y DropBear se llevan a cabo del mismo modo que en test previos: todas las sesiones SSH

[121] "Kernel /proc/sys/net/netfilter/nf_conntrack_* Variables", *Kernel.org*, 2019, https://www.kernel.org/doc/Documentation/networking/nf_conntrack-sysctl.txt

[122] "Kernel /proc/sys/net/ipv4/* Variables" *Kernel.org*, 2020, https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt

deban estar establecidas y operativas antes de iniciar los test propiamente dichos. Esto incide en el procedimiento seguido para medir los tiempos y cuándo deben empezar a computarse. El motivo por el que se los test se planifican de este modo radica en la característica funcional que tiene el esquema de comunicación presentado, basado en desacoplo de seguridad que presenta el establecimiento de sesiones persistentes del dispositivo IoT hacia el servidor. Por consiguiente, cuando se efectúa una petición HTTP por parte del dispositivo IoT, no hay necesidad de establecer ninguna sesión SSH o proxificación, ya que se encuentran previamente establecidas (la misma situación sería aplicable si, desde el servidor, la plataforma necesitase contactar con un dispositivo IoT). Esto se considera una ventaja importante del sistema de comunicación IoTsafe propuesto.

Cada serie de test se realiza 25 veces y los tiempos requeridos para cada serie se promedian. Los resultados de estas series de test se muestran en la Fig. 4.4-7, usando escalas no lineales en ambos ejes.

El comportamiento lineal observado para cada serie de test con respecto a la cantidad de peticiones HTTP por segundo procesables en paralelo, muestran una tasa estable de RPS procesada por el servidor IoT. Si se llegara a aumentar la cantidad de peticiones, algunas de esas conexiones deberían descartarse y la RPS efectiva disminuiría debido a la congestión. Las peticiones HTTP sin seguridad se representan como referencia.

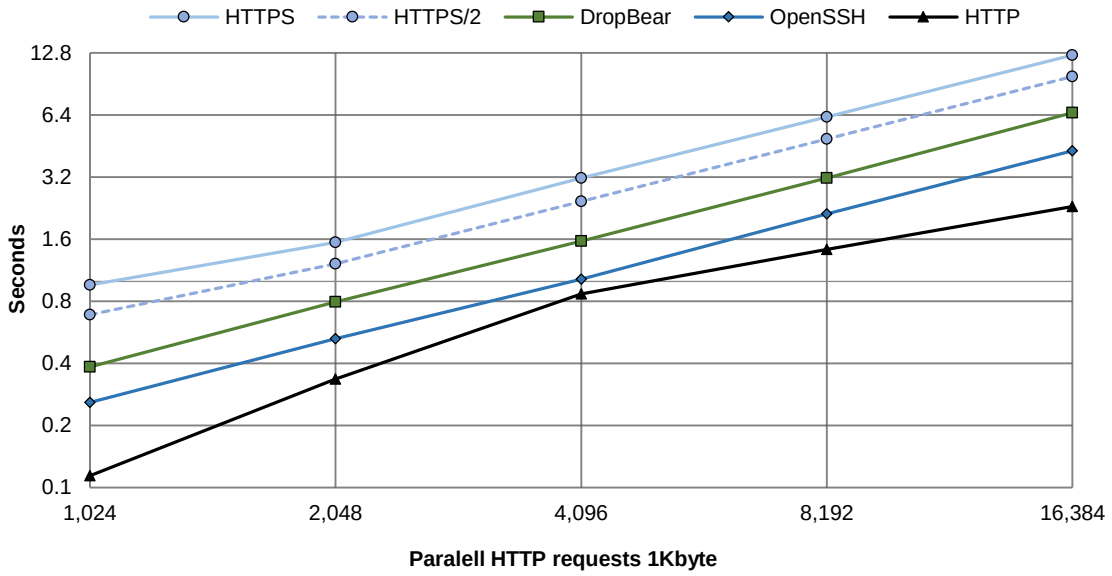


Fig. 4.4-7: Tiempo requerido para realizar diferentes peticiones HTTP de 1 Kbyte de payload utilizando los protocolos y configuraciones estudiados. Las proxificaciones SSH se establecen utilizando los valores optimizados de P y S.

Los resultados obtenidos se resumen en la Tabla VII. En ella se muestra el rendimiento en términos de RPS. De las cifras obtenidas se puede deducir que ambas implementaciones de SSH

proporcionan una significativa mejora en rendimiento con respecto a HTTPS y HTTPS/2. OpenSSH, por ejemplo, supera a HTTPS/2 por un factor de 2 en términos de RPS y prácticamente por un factor de 3 respecto a HTTPS.

TABLA VII
PROMEDIO DE PETICIONES POR SEGUNDO (RPS) Y MEJORA DE RENDIMIENTO (ΔP)

Caso	Peticiones por segundo (RPS)	Mejora de rendimiento (ΔP) vs. HTTPS	Mejora de rendimiento (ΔP) vs. HTTPS/2
HTTPS	1,297	0 %	-22%
HTTPS/2	1,665	28 %	0%
DropBear	2,542	96 %	53%
OpenSSH	3,857	197 %	132%
HTTP	6,284	-	-

4.4.5. **Análisis de los resultados**

El análisis del rendimiento de los servidores necesarios para implementar este esquema de comunicación en los dispositivos demuestra que resulta factible y escalable, utilizando un esquema en tres niveles: dispositivo IoT, *gateway* IoT, server IoT. En el esquema propuesto, cada *gateway* IoT puede albergar y comunicarse simultáneamente con más de 1.000 dispositivos IoT diferentes manteniendo un consumo de RAM de unos 5GB, utilizando la versión *debug stand-alone* (sin optimizar) de DropBear. En todo caso, las comunicaciones agregadas de los *gateway* de los dispositivos IoT pueden enviarse de forma muy eficiente hacia el servidor IoT utilizando funcionalidades de multiplexación disponibles en SSH. De ese modo, se ha demostrado que con las implementaciones estándar de SSH y con un servidor virtual en la nube de 16Gb de RAM y 6 núcleos virtuales se pueden llegar a establecer más de 262.000 proxificaciones, lo que se traduce a 262.000 dispositivos, si cada uno de ellos requiere únicamente de una proxificación. Esta implementación puede alcanzar una ratio de 2.542 RPS con la versión *stand-alone* de Dropbear, pudiendo mejorar hasta los 3.857 RPS con OpenSSH, en ambos casos manejando peticiones HTTP de 1 Kbyte aseguradas por proxificación.

Dado que la cifra de RPS se encuentra bastante por debajo del número de dispositivos manejables por un único servidor, la solución de comunicación propuesta, tal y como se presenta, resulta técnicamente de interés: si se requiere aumentar los RPS, se podría optar por contar con servidores que gestionen menos dispositivos IoT cada uno. También se puede estudiar en todo caso la optimización de la implementación del protocolo SSH, retirando funcionalidades no necesarias, y reduciendo así el consumo de RAM por proxificación en el servidor. Esto permitiría desplegar nodos IoTsafe de comunicación que concentrasen un número mayor de proxificaciones.

A este respecto, también se estudiaron las limitaciones actuales del *framework* I/O basado en multiplexación y que se utiliza en ambas implementaciones para manejar los *socket,* y se propone sustituirlo por otro basado en señales emitidas por el *kernel,* que resulta mucho más eficiente. Dicho *framework* lo utilizan por defecto aplicaciones compiladas con el lenguaje Golang (utilizando *epoll()*[123] *en lugar de select()*). Este lenguaje proporciona un rendimiento muy superior en operaciones de manejo y monitorización de descriptores de ficheros y *socket,* y actualmente existen iniciativas[124] que pretenden desarrollar SSH en este lenguaje, pudiéndose mejorar drásticamente en un futuro las implementaciones SSH orientadas a comunicaciones basadas en el desacoplo de seguridad.

En todo caso, las implementaciones de IoTsafe que se pretendan desplegar actualmente resultan perfectamente factibles con los desarrollos SSH actuales y, en un futuro próximo, pueden actualizarse con aquellas que utilicen GoLang o cualquier otra tecnología, gracias a la modularidad del esquema de comunicación y la independencia entre los módulos de *software* derivada de la política de desacoplo de seguridad. De ese modo, únicamente se deberá sustituir un módulo de seguridad por otro sin tener que modificar ningún otro binario del dispositivo IoT. Esto permite extender la vida útil de los dispositivos IoT y reducir costes de mantenimiento derivados de las actualizaciones por motivos de seguridad.

[123] "netpoll_epoll.go – epoll GoLang programming for ready network connections", *Github*, 2017, https://github.com/golang/go/blob/91c9b0d568e41449f26858d88eb2fd085eaf306d/src/runtime/netpoll_epoll.go
[124] "SSH package for GoLang", *GoDoc*, https://godoc.org/golang.org/x/crypto/ssh

5. CONCLUSIONES Y TRABAJOS FUTUROS

5.1. CONCLUSIONES

Los dispositivos IoT y su arquitectura se encuentran en constante crecimiento y evolución. Fruto de ello, tecnologías de carácter vertical de tipo *ad-hoc* son las predominantes actualmente en los ecosistemas IoT, para dar solución a problemas específicos de una forma ágil y competitiva. Por otro lado, las sinergias entre verticals pueden propiciar la aparición de nuevas oportunidades de negocio, pero la conectividad entre ellos normalmente resulta compleja. Además, la seguridad y privacidad están cobrando una relevancia especial, no sólo por parte de los desarrolladores, sino principalmente por los cuerpos reguladores internacionales.

En este trabajo se ha analizado el uso del desacoplo de seguridad para entornos IoT, y se ha propuesto un sistema de comunicación que emplea este desacoplo de manera segura y escalable en *gateway* y servidores. Se han estudiado técnicas en las que la seguridad de los dispositivos IoT se desacopla de sus aplicaciones, para evitar que el desarrollador y el usuario final de los dispositivos IoT tengan que estar al tanto de las últimas actualizaciones de seguridad.

En la tesis se ha elaborado una propuesta técnica para implementar este desacople de seguridad. Esta propuesta cristaliza en un método y sistema de comunicación por proxificación denominado IoTsafe. El esquema propuesto simplificará los requisitos y la complejidad del diseño del *software*, permitiendo delegar las funciones de seguridad a módulos independientes, pudiendo ser gestionados externamente por empresas con mayor experiencia en seguridad. Todas estas medidas acaban contribuyendo en su conjunto a un ecosistema IoT más saludable:

- **Seguridad mejorada del dispositivo IoT**: protege a los dispositivos IoT de múltiples amenazas, al crear un límite bien definido entre las aplicaciones IoT y los módulos stand-alone de seguridad para las comunicaciones. En particular, esta estrategia protege de la piratería masiva de dispositivos, una preocupación principal en IoT.
- **Diseño y desarrollo mejorados**: se reducen los tiempos de los ciclos de diseño, al desacoplar los problemas de seguridad y permitir la reutilización de componentes de seguridad especializados en otros proyectos IoT.
- **Certificación mejorada**: la utilización de IoTsafe podrá simplificar los procesos de certificación y auditoría de seguridad de los dispositivos y sistemas IoT. Esto se conseguiría gracias a la separación de las funcionalidades de las comunicaciones seguras, de las aplicaciones IoT, para que estas se produzcan de forma transparente y sin haber requerido condicionantes especiales durante el desarrollo de dichas aplicaciones (ningún *framework* concreto, protocolo de seguridad, etc.). Los aspectos a certificar son concretos y transversales a cualquier dispositivo: los módulos de seguridad y procedimientos de gestión y mantenimiento de los mismos.

- **Incremento de la vida útil de los dispositivos IoT**: IoTsafe permitirá que las comunicaciones de los dispositivos IoT se realicen correctamente y por mayor tiempo, sin necesidad de actualizaciones específicas del tipo de dispositivo o aplicación IoT, que involucre a los equipos de desarrollo originales. La posible externalización de la gestión de los aspectos de seguridad garantiza este punto.
- **Horizontalización de las comunicaciones:** IoTsafe sirve como base para un *framework* de transporte que permita extrapolar las ventajas de seguridad conseguidas para los dispositivos IoT, también a los *gateway* y servidores. Su esquema de comunicación basada en contextos de seguridad y "circuitos virtuales" transparentes a las aplicaciones finales, permite simplificar los *middleware*, proporcionar una mayor garantía de compatibilidad entre *verticals* y diferenciar el trato de la seguridad, también en estos *middleware,* del mismo modo que ocurre en los dispositivos IoT.

Este esquema de comunicación permitirá que los dispositivos IoT cumplan más fácilmente con las regulaciones y alivien las preocupaciones de los desarrolladores y usuarios sobre la seguridad. Además, los mecanismos de IoTsafe son genéricos y aplicables a gran variedad de desarrollos IoT. IoTsafe aspira a proporcionar tecnología que ayude de manera fundamental al diseño seguro de dispositivos IoT y permita su uso más amplio, facilitando la creación de un Mercado Único Digital confiable.

Las diversas pruebas realizadas y los resultados obtenidos en este trabajo avalan el uso de IoTsafe con herramientas *software* comunes en una gran variedad de dispositivos IoT hoy en día. Se ha demostrado que los dispositivos basados en Linux, o en algún sistema equivalente POSIX que cuente con SSH, son capaces de incorporarse rápidamente a un servidor IoTsafe. Por su parte, también se ha documentado la existencia de alternativas para dispositivos que usen RTOS y puedan ser capaces de incluir WolfSSH en su desarrollo.

Los resultados ofrecidos sirven para definir las principales ventajas para los dispositivos que incorporan IoTsafe. Se ha logrado un funcionamiento satisfactorio en entornos móviles utilizando *hardware* y *software* de propósito general. Las comunicaciones proxificadas no suponen un incremento apreciable en latencia u *overhead* y la estabilidad de las comunicaciones es similar a la encontrada con estrategias tradicionales. En el caso de emplear una interfaz de red 802.15.4 con funcionalidades más restringidas, se obtuvo igualmente un resultado favorable, incluso usando protocolos de aplicación de alto nivel del tipo HTTPS o HTTPS/2:

- En las comparaciones realizadas con HTTP y HTTP2, las comunicaciones proxificadas empleando SSH pueden llegar a ofrecer mejor *goodput* para cualquier tamaño de *payload* HTTP (Fig. 4.2-3). A partir de peticiones de 4Kbytes, esta mejora de *goodput* es del doble frente a las soluciones basadas en TLS.
- Respecto al consumo de energía de CPU, la solución basada en SSH mejora a la tradicional de TLS con *payload* de peticiones HTTP a partir de los 16 Kbytes (Fig. 4.2-4). Dicho consumo de energía se reduce más de un 40% a partir de 66 Kbytes.

- Respecto al consumo de energía en la interfaz wireless usada, el requerido por la solución SSH es menor que el de el de TLS, a partir de peticiones HTTP con *payloads* de 1 Kbyte. De los 4 Kbytes en adelante, el consumo con SSH cae a más de la mitad. (Fig. 4.2-5).

También se validó el uso de CoAP como protocolo de capacidades restringidas, empleando la interfaz de red 802.15.4. En este caso, se estudió el efecto que tiene en las comunicaciones un canal con interferencias reales. Los resultados demostraron que las comunicaciones basadas en proxificaciones SSH superaron a las de CoAP con una mejora del rendimiento de hasta un 63% (TABLA IV), gracias a un mejor control de congestión que el utilizado en el caso de CoAP. Esto permite manejar de forma más efectiva las interferencias en el canal real que la implementación estándar que tiene CoAP.

Por otro lado, también se analizaron y resolvieron adecuadamente las posibles limitaciones inicialmente identificadas en el servidor:

- El consumo de RAM por proxificación manejada en un servidor IoT resultaba inicialmente demasiado elevado para poder garantizar un escalado adecuado de los proyectos.
- Se propuso un esquema de tres niveles, similar al que se sigue en multitud de proyectos IoT: dispositivos, *gateway*, servidores. Esto permite multiplexar las comunicaciones de dispositivos IoT locales en los *gateway* que los gestionan directamente, pudiendo llegar a gestionar en un servidor estándar más de 260.000 proxificaciones (Intel(R) Xeon(R) CPU E5-2680 v3 2.50 GHz, 6 hilos y 16 GB de RAM).
- El rendimiento medido en el servidor empleado llegó hasta los 3.857 RPS con peticiones HTTP sobre SSH de 1´Kbyte. Esta marca supone un incremento neto del 132% en el rendimiento frente a HTTPS/2, y del 197% frente a HTTPS, superando a HTTPS/2 por un factor de más de 2 en términos de RPS y respecto a HTTPS por un factor cercano a 3. Estos resultados concluyen que el esquema propuesto por IoTsafe supone además una mejora notable en el rendimiento de las comunicaciones seguras en el servidor.
- Los análisis realizados permiten además pensar que los resultados obtenidos podrían ser mejorados:
    - Optimizando OpenSSH o DropBear en el servidor para ocupar menos memoria RAM, eliminando funcionalidades no necesarias.
    - Desarrollando una versión de SSH en GoLang que utilice *epoll()* para manejar los descriptores abiertos de *socket*.

5.2. TRABAJOS FUTUROS

Como trabajo futuro se plantea continuar desarrollando IoTsafe y comercializar su concepto mediante licencia de explotación de patentes [110]. Por otro lado, también se ha contemplado la idea de elaborar una solución más ambiciosa que basándose en las ideas de IoTsafe aspire a

convertirse el un framework de red más completo que cumpla con todos los aspectos analizados en la sección 3.4.4. Este tipo de desarrollo, podría contribuir en las soluciones de diversas inquietudes como las planteadas en la convocatoria H2020 SU-DS02-2020 "*Intelligent security and privacy management: Boosting the effectiveness of the Security Union (SU)*"[125]. Este proyecto incluirá, junto a IoTsafe, otras iniciativas de envergadura, complementarias entre sí, que permitan en su conjunto contribuir en la búsqueda de una solución global a los problemas de seguridad planteados en la convocatoria tal y como se presenta en la Fig. 5.2-1.

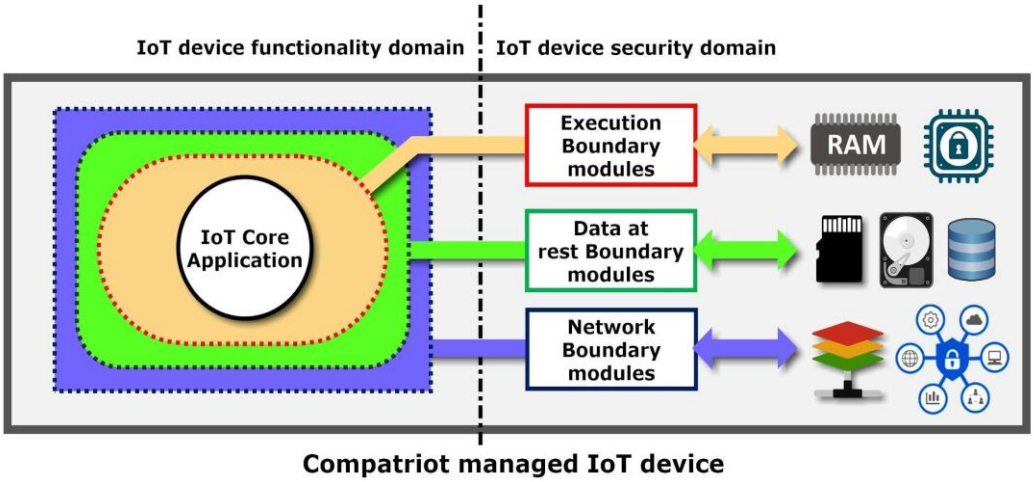


Fig. 5.2-1: Propuesta del esquema de desacoplo de seguridad aplicado en tres niveles propuesto en el que IoTsafe se ocupa del nivel de red. Los módulos de seguridad involucrados funcionan de forma independiente a la aplicación principal permitiendo que las actualizaciones de seguridad de dichos módulos y sus procesos de certificación sean más sencillos e independientes de la aplicación central del dispositivo IoT.

En dicho proyecto se prevee incluir tres iniciativas principales para cumplir con los objetivos planteados por la Comisión en la convocatoria anteriormente mencionada:

El ámbito de red es tratado por IoTsafe y todo lo presentado en esta tesis se aplicaría en el desarrollo del proyecto, teniendo siempre en cuenta los otros dos ámbitos de acción.

El ámbito de ejecución se centra en tratar de garantizar un entorno de ejecución seguro para las aplicaciones IoT de forma transparente (*chain of trust*[126]). IoTsafe se limita originalmente a abordar el desacoplo de la seguridad de las aplicaciones y su gestión en el servidor. Esta estrategia de IoTsafe más simplista requiere de un sistema *hardware/software* que sea seguro on ciertas garantías. La "confianza" necesaria para una correcta operación del dispositivo se entiende como una certeza de que las distintas partes que lo conforman son igualmente de confiables. Además, se requiere de un control y conocimiento del funcionamiento interno de cada parte del dispositivo para garantizar que puedan considerarse como segurid y de confianza. Sin

[125] "*Intelligent security and privacy management: Boosting the effectiveness of the Security Union (SU)*", SU-DS02-2020, https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/su-ds02-2020. Deadline 27 agosto 2020.

[126] Why Trusted Execution Environment and Security by Separation on IoT Edge Devices Are Important?, 2019, https://iot.ieee.org/newsletter/september-2019/why-trusted-execution-environment-and-security-by-separation-on-iot-edge-devices-are-important

embargo, en software constituido por aplicaciones muy complejas con múltiples proveedores de *software*, librerías, controladores, etc., establecer ese grado de confianza puede resultar muy complicado (y costoso), dado que el desarrollador de la aplicación no ha implementado cada módulo, o incluso desconoce totalmente los detalles del funcionamiento interno de dichas partes, validando su funcionamiento únicamente como "*black-boxes*" [167].

El ámbito de los datos, también presta atención a los aspectos de seguridad y de privacidad, que cobran una especial relevancia si se contemplan desde las normativas de protección de datos recientes y, el especial tratamiento que esta información requiere ya de forma obligatoria.

Los tres ámbitos presentados suponen una forma novedosa de abordar el desarrollo de *software* y de los dispositivos, que tiene una implicación directa en las metodologías de certificación y auditoría. En una pubicación reciente de ENISA relativa a los estándares y seguridad en IoT [122], se describen una serie de recomendaciones para estandarizar procedimientos relacionados con la seguridad de los dispositivos IoT y sus aplicaciones. En particular, en su Anexo B se describe un esquema muy similar al que se planteará por nuestro proyecto y en el que se basa IoTsafe: modularidad e independencia entre elementos para permitir "evaluaciones compuestas". Este concepto se fundamenta en la flexibilidad que proporcionan la serie de estándares ISO/IEC 15408[127] que basa la certificación del producto final en la certificación de las partes del mismo al considerarlo un "*composed assurance package*"[128].

La aplicación de evaluaciones compuestas puede simplificar y economizar de forma notable los procesos de certificación de los dispositivos. Para ello es necesaria la certificación y evaluación de los procedimientos de fabricación y diseño de algunas partes esenciales del mismo que fundamenten el "*root of trust*" [168]. El módulo que sustenta esta confianza ser lo más pequeño posible y limitar su complejidad a lo estrictamente indispensable. Al reducir su tamaño y complejidad, resulta más sencillo de certificar y más difícil de comprometer para cargar malware en él [168]. A su vez, la *chain of trust* puede utilizarse para garantizar que la aplicación que se va a cargar es de confianza, y esto puede realizarse de forma transparente a la propia aplicación.

En definitiva, se espera que este proyecto sea capaz de producir soluciones altamente innovadoras para la industria, convirtiendo el conocimiento en productos y servicios comercializables, al unir la investigación y la innovación[129].

[127] "ISO/IEC 15408-1:2009 Information technology — Security techniques — Evaluation criteria for IT security — Part 1: Introduction and general model", *ISO Online Browsing Platform (OBP)*, 2009, https://www.iso.org/obp/ui/#iso:std:iso-iec:15408:-1:ed-3:v2:en

[128] "Common Criteria for Information Technology Security Evaluation. Part 3: Security assurance components" *Common Criteria Portal*, 2009, https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R3.pdf

[129] *European Commission, Key Enabling Technologies*, "*Challenges for Europe*," https://ec.europa.eu/growth/industry/policy/key-enabling-technologies/challenges_en

ANEXOS

ANEXO A: PROTECCIÓN DEL POOL DE DIRECCIONES RESERVADAS

Este anexo explica el concepto el concepto de “protección del pool de direcciones reservadas”, que consiste en que un conjunto de direcciones IP locales del dispositivo_A queda reservado en el perfil_A específicamente para este sistema. Estas direcciones son protegidas al arrancar el sistema operativo del dispositivo_A por el *framework* de red, a instancias del servidor *proxy-firewall,* por medio de las siguientes reglas generales:

- Una regla general de marcado de paquetes que determine eliminar cualquier posible marcado de paquetes de red en cualquier comunicación, producida entre *socket* locales del dispositivo_A, con direcciones locales pertenecientes al pool de direcciones locales reservadas.
- Una regla general de filtrado de paquetes que determine descartar cualquier paquete con origen y destino un *socket* local del dispositivo_A que carezca de un marcado, siempre y cuando alguno de los *socket* utilice una dirección de red local perteneciente al pool de direcciones locales reservado.
- Una regla general de filtrado de paquetes que determine descartar cualquier paquete con origen o destino una dirección local del pool de direcciones locales reservado para este método, que tenga como origen o destino una dirección no local.

Además, se debe realizar la supresión, durante la secuencia de arranque del sistema operativo del dispositivo_A, de cada registro_acceso y permiso_A_U de la base de datos relativo a cada registro_*socket* cuyo tipo_*socket* sea *socket-proxy*, y la supresión del propio registro_*socket*, ya que esos registros_acceso y registros_*socket* son relativos a proxificaciones entre dispositivos_A y son considerados temporales.

ANEXO B: MÉTODO DE COMUNICACIÓN SEGURA POR PROXIFICACIÓN DE *SOCKET* DE RED

En este anexo se expone en detalle el método que permite realizar comunicaciones seguras entre procesos por medio de proxificaciones de *socket* de red [110]. Para ello, inicialmente se establecen los *socket* protegidos en el dispositivo_B y en el dispositivo_A, lo que comprende las siguientes acciones:

- Durante el arranque del dispositivo_A, el servidor *proxy-firewall* ingresa unas reglas de marcado y filtrado a través del *framework* de red al sistema operativo del dispositivo_A, para proteger el pool de direcciones asignado en el perfil_A. Estas reglas de carácter general comprenden:
  - Eliminar cualquier posible marcado de paquetes de red generado por los procesos en cualquier comunicación, producida entre *sockets* del dispositivo_A, con direcciones de red pertenecientes al pool de direcciones locales reservado para este método.
  - Descartar cualquier paquete con origen y destino un *socket* local del dispositivo_A que carezca de un marcado, siempre y cuando alguno de los *socket* utilice una dirección de red local perteneciente al pool de direcciones reservado para este método.
  - Durante el arranque del dispositivo_B, el agente_B ingresa al menos, una regla al *framework* de red del sistema operativo del dispositivo_B para proteger cada *socket_B* definido en el perfil_B_A. Esta regla de carácter general consiste en descartar cualquier paquete de red que tenga una dirección origen no local y cuyo destino sea alguno de los *socket_B*.
  - Ingresar, durante la secuencia de arranque del sistema operativo del dispositivo_A y a través de un *framework* de red de este último, una regla general de filtrado de paquetes por parte del servidor *proxy-firewall,* que determine descartar cualquier paquete con origen o destino una dirección local del pool de direcciones locales reservado para este método, que tenga como origen o destino una dirección no local.

Una vez se han establecido las reglas de protección del pool de direcciones en el dispositivo_A, se registran en la base de datos los *socket_A* que aún no estén registrados, y se aplican las reglas de marcado pertinentes que definan los contextos de seguridad amplios que correspondan en el dispositivo_A, para conceder acceso permanente a los grupos de usuario del dispositivo_A que se definan en el perfil_A. Todo ello comprende de las siguientes acciones:

- El ingreso, si no existe, durante la secuencia de arranque del sistema operativo del dispositivo_A, de un registro_*socket* en la base de datos del dispositivo_A relativo a cada

socket_A del dispositivo_A recogido en el perfil_A conteniendo cada registro_*socket.* Esta información comprende:

  - El id_DispSocket, que en este caso sería el id_A.
  - El id_Disp*Proxy*, que en este caso sería también igual al id_A.
  - El alias, que en este caso sería igual al empleado por el *socket_A* a registrar.
  - El *socket*, que en este caso sería igual al *socket* local utilizado por el proceso_A para el *socket_A* definido en el perfil_A.
  - El tipo_*socket*, que en este caso sería igual a "*socket_A*".

- El ingreso, si no existen, de los permisos_A_G y de cada registro_acceso_A, tal y como se contemplen en el perfil_A.
- El ingreso de unas reglas de marcado prioritarias respectivas a cada registro_acceso_A de cada *socket_A,* concedidas permanentemente a los grupos de usuarios del dispositivo_A, según esté definida cada una en forma de permiso en la base de datos del *connection-broker*.

Tras esto, el servidor *proxy-firewall* verifica a través del *connection-broker* que en la base de datos no existan accesos en vigor relativos a registros_*socket* de tipo *socket-proxy*. En caso de existir, borrará cada registro_acceso asociado a dicho registro_*socket,* incluyendo el propio registro_*socket*. Estos registros, relativos a proxificaciones entre el dispositivo_A y dispositivos_AF son considerados temporales y durante el inicio del sistema operativo del dispositivo_A deben suprimirse.

El proceso continúa cuando el servidor *proxy-firewall* del dispositivo_A recibe una petición de establecimiento de sesión del agente_B para realizar unas proxificaciones, todo lo cual comprende las siguientes acciones:

- Establecer una sesión de comunicaciones segura que permita efectuar proxificaciones de *socket* entre el dispositivo_A y el dispositivo_B, donde el primero utilice un servidor *proxy-firewall* que incluya, y el segundo utilice un agente_B que inicie el establecimiento de dicha sesión, empleando un cliente *proxy stand-alone* que implemente un protocolo de comunicación segura, usando unos datos de configuración y credenciales del usuario_A_B definidos por su perfil_B_A. De este modo, el dispositivo_B establece la sesión con el dispositivo_A utilizando la cuenta usuario_A_B.
- Proceder con el establecimiento de las proxificaciones directas. Éstas permiten acceder a cada *socket_A* requerido desde el dispositivo_B. Dichas proxificaciones las solicita el agente_B según el perfil_B_A y quedan protegidas por el protocolo seguro utilizado en la sesión de comunicaciones entre el agente_B y el servidor *socket-proxy*.

- Establecer las proxificaciones reversas. Éstas permiten que cada *socket_B* sea accesible por las aplicaciones del dispositivo_A de forma local, siempre y cuando el usuario del sistema operativo que las ejecuta tenga permiso para ello. Cada *socket_B* que debe proxificarse está definido también en el perfil_B_A.
- Registrar las proxificaciones reversas en una base de datos de un *connection-broker* del dispositivo_A, generando un registro_*socket* por cada una de ellas.

Una vez seguido este método, los procesos del dispositivo_B pueden acceder a cada *socket_A* proxificado de forma local, y los procesos del dispositivo_A que dispongan de permiso_A_U pueden acceder a los *socket-proxy* reversos, una vez soliciten la resolución al servidor *proxy-firewall* del dispositivo_A y se configuren los contextos de seguridad estrictos pertinentes.

ANEXO C: EJEMPLOS DE IMPLEMENTACIÓN DE IOTSAFE

En este anexo se incluyen 10 ejemplos de implementación de IoTsafe que tratan de ilustrar la operación de distintas funcionalidades que puede soportar. Estos ejemplos suponen una forma práctica y concreta de mostrar una configuración adecuada de los distintos elementos que conforman IoTsafe y de su forma de interacutar. El texto está extraído de la solicitud de patente presentada por el autor en [110].

Ejemplo 1: Establecimiento de una proxificación directa de un *socket_A* en un dispositivo_B y de una proxificación reversa de un *socket_B* en un dispositivo_A

En la Fig. C-1, se describe un diagrama de secuencia en el que, en una realización de IoTsafe, el dispositivo_B requiere proxificar dos puertos: un *socket_B* de forma directa y un *socket_A* de forma reversa. En este ejemplo intervienen un dispositivo_B con un agente_B (201) y un dispositivo_A con un servidor *proxy-firewall* (202), un *connection-broker* (203) y una base de datos (204). Este ejemplo es generalizable para establecer cualquier número de proxificaciones directas y/o inversas en la misma sesión, sin necesidad de realizarse recurrentemente.

En primer lugar, el agente_B inicia la sesión (205) en el servidor *proxy-firewall,* utilizando la configuración definida en un perfil_B_A. Durante el proceso de autenticación (206), el servidor *proxy-firewall* determina el número y tipo de proxificaciones que el agente_B puede realizar a partir del perfil_B_A usado en el establecimiento de sesión. Una vez se realiza la autenticación y se establece la sesión, el servidor *proxy-firewall* realiza un *Fork* de sí mismo (207) en la cuenta usuario_A_B. De este modo, se genera una nueva instancia del servidor *proxy-firewall* como *Fork* (208) en esa cuenta, lo que permite una separación de privilegios. Este *Fork* solicita (209) al *connection-broker* la reserva de un *socket* libre para establecer el *socket-proxy* reverso para el *socket_B*. Así mismo, dicha solicitud también contiene un requerimiento de resolución del *socket_A* para obtener el *socket* protegido asignado a dicho *socket_A*.

El *connection-broker* consulta (210) en la base de datos si existe el permiso_A_G para el *socket_A* e inserta o actualiza el registro_*socket* para el *socket_B* según el perfil_B_A. Si este registro_*socket* existía previamente, se verifica si también existe algún registro_acceso en vigor relativo a ese registro_*socket*. En caso de existir, se mantiene el valor de *socket* asignado en el registro_*socket*. Si no existe, el valor de *socket* para el registro_*socket* se actualiza con otro nuevo que sea local; perteneciente siempre y en todo caso al pool de direcciones reservadas para este método y que el *socket* se encuentre disponible.

En caso de no existir el registro_*socket* o en el caso de tener que generar un registro_acceso completo, el *socket* se elige aleatoriamente entre los que se encuentren libres en el pool de direcciones reservadas. Una vez terminadas estas acciones, se procede con la respuesta (211). Esta respuesta incluye la resolución del *socket-proxy* (el valor del *socket* asignado en el registro_*socket* para el *socket_B*) y la resolución del *socket_A* (el valor asignado en el registro_*socket* relativo al *socket_A*) en forma de registro_*sockets*. Tras esto, se notifica al *Fork* de ello (212) y éste a su vez hace lo propio con el agente_B (213). Éste último, cursa una solicitud (215) al *Fork* para que inicie un *socket* protegido en escucha que corresponda al *socket-proxy* reverso (215) previamente asignado. Así mismo, el agente_B inicia en escucha un *socket* local protegido del dispositivo_B que corresponde al *socket-proxy* directo (216).

Tras realizarse la proxificación, el servidor *proxy-firewall* monitoriza periódicamente que se encuentren operativa. De lo contrario, inicia el procedimiento de cierre para su restablecimiento, como puede apreciarse en la Fig. C-2.

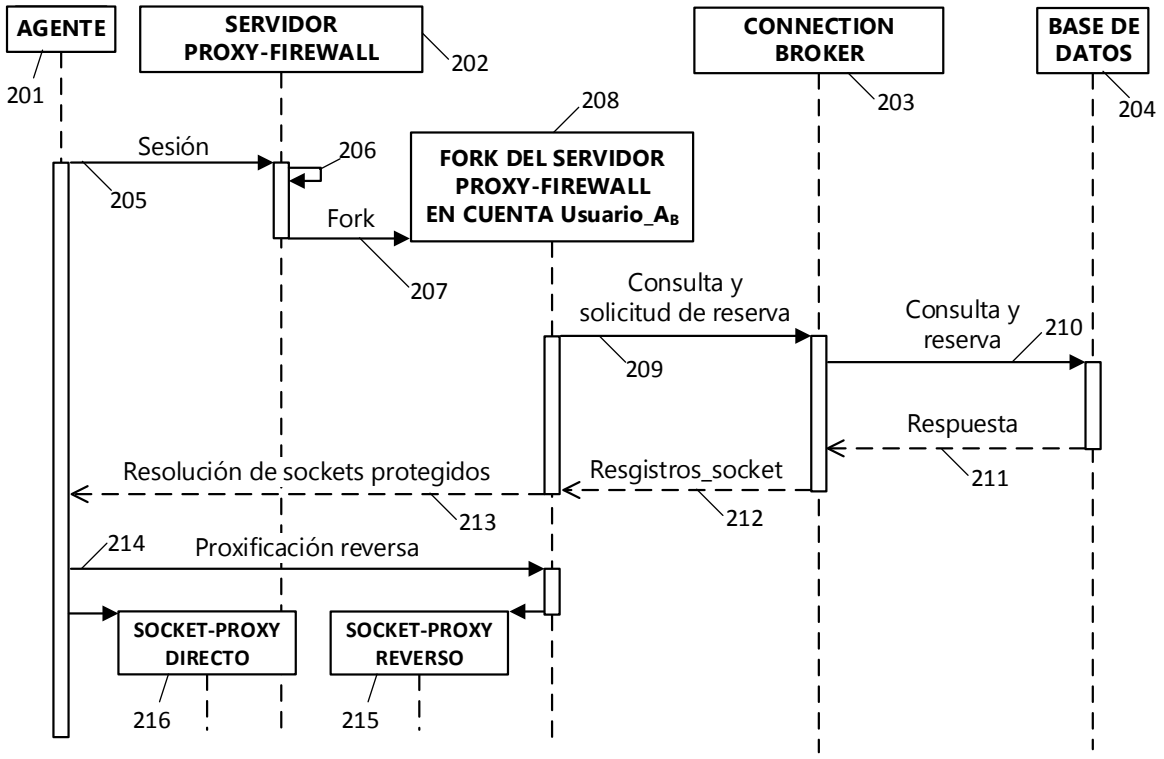


Fig. C-1: Diagrama de secuencia del establecimiento de un socket-proxy directo y reverso.

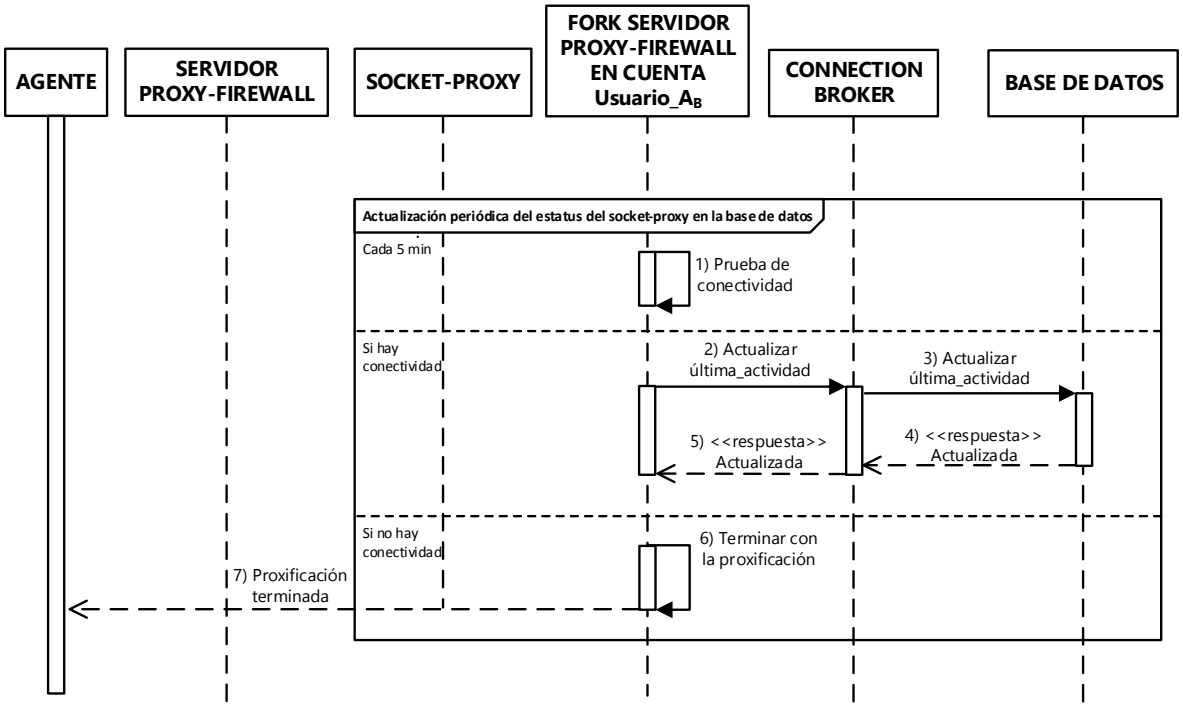


Fig. C-2: Diagrama de secuencia de la actualización del valor "ultima_actividad" en el registro_socket de una proxificación.

En una implementación basada en SSH, SSHD ejecuta obligatoriamente un comando especificado en el fichero *authorized_keys*[130] por cada *login* exitoso de cada usuario (clave privada). Este comando, ejecuta un *script* para que el sistema verifique los parámetros de conexión del usuario y valide la petición antes de proceder. Se lleva a cabo mediante un *wrapper* autorizado por */etc/sudoers*[131]. Esto permite que, manteniendo una separación de privilegios, se pueda realizar una llamada al sistema desde la cuenta de usuario empleada, pero sin posibilidad de cambiar datos relativos a la solicitud de conexión, tal y como se puede apreciar en la figura Fig. C-3, en la que se produce la siguiente correspondencia de *software*:

- *Fork* servidor *proxy-firewall* en cuenta Usuario_A_B:
  - SSHD *Fork*, Connection_*script*.sh, Fireshell_wrapper.sh
- *Connection broker*: TesPX.php
- *Network Framework*: Fireshell.sh
- Usuario_A *domain*: IoT usr plf *domain* (Cuenta de usuario de un dispositivo IoT conectado al servidor.)

[130] "SSH Server-Side Client Files, authorized_keys," *Wikibooks,* https://en.wikibooks.org/wiki/OpenSSH/Client_Configuration_Files#~/.ssh/authorized_keys
[131] "sudoers - default sudo security policy module", *Linux man pages*, 2012 https://linux.die.net/man/5/sudoers

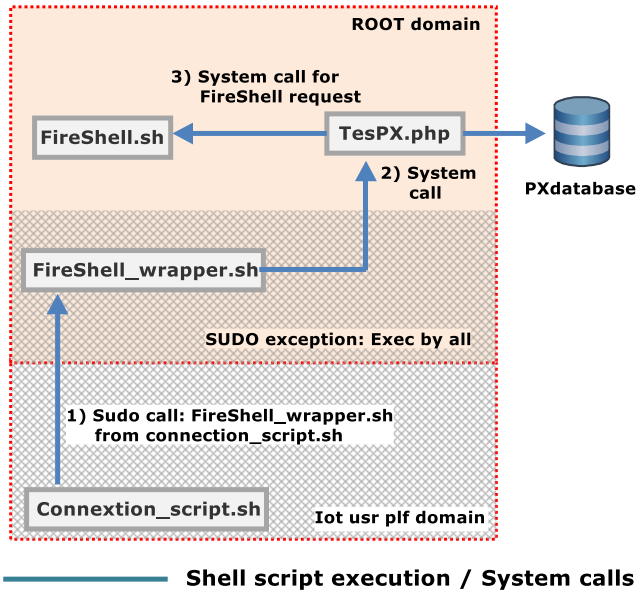


Fig. C-3: ***Figura conceptual representando la separación de privilegios y la ejecución de "connection_script.sh" por la instancia SSH creada tras un login exitoso para poder validar la solicitud de proxificación. FireShell_wrapper.sh es capaz de identificar el usuario de sistema origen de la llamada y TesPX.php valida sus permisos antes de habilitar los contextos de seguridad.***

En la Fig. C-4 se muestra el diagrama de secuencia seguido por una implementación del ejemplo que estamos presentando, en el que se utiliza la siguiente correspondencia de *software*:

- El Agente y el servidor *proxy stand-alone* (parte del servidor *proxy-firewall*) son ejecutados mediante una implementación estándar de OpenSSH
- El *Fork* del servidor *socket-proxy* en la cuenta de usuario_A_B del dispositivo_A corresponde con la instancia que SSHD realiza cuando se genera una conexión.
- Servidor *proxy stand-alone*: SecurePX (sus *scripts* y SSHD).
- Connection_broker: TesPX.
- *Network Framework*: Fireshell y PortPX
- Monitor de accesos: Una parte del connection_*script*.sh.

En este ejemplo, la validación de los parámetros de conexión (los *socket* a proxificar) se realiza posteriormente al establecimiento de dicha conexión por la ejecución del connection_*script*.sh. Dichos puertos se validan y se habilitan en el *authorized_keys* para esa clave.

En una implementación directa de dicho método, el proceso completo de conexión se realiza en dos fases. En la primera se validan los parámetros y se actualiza el *authorized_keys* para permitir las proxificaciones, y en la segunda se realizan dichas proxificaciones. Sin embargo, una implementación más óptima, incluiría estos procesos en la implementación de SSH, particularmente una validación previa a la apertura de cualquier proxificación.

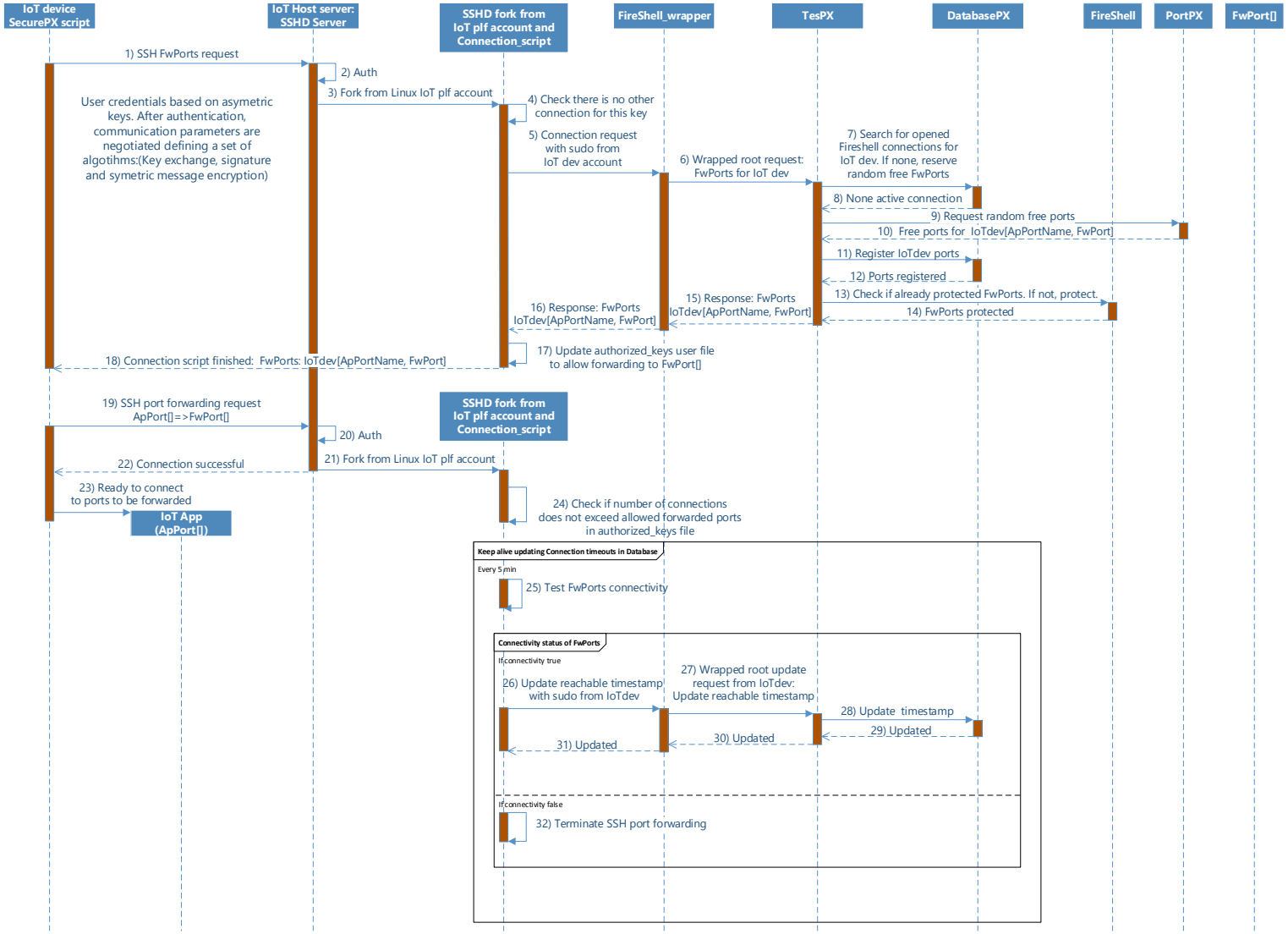


Fig. C-4: Diagrama de secuencia de un ejemplo completo de establecimiento de proxificación usando SSH.

E1.1. Separación de privilegios en el Servidor

La implementación descrita a continuación parte del hecho de utilizar *software* estándar usado en sistemas en producción, que no requiera de parches o modificaciones para obtener una versión segura y funcional. En todo caso, también se señalan los aspectos que, en un futuro, se podrían modificar para conseguir un funcionamiento óptimo.

Las dos principales implementaciones de SSH para Linux disponen de funcionalidades útiles que conviene considerar: OpenSSY y DropBear. Esta última es una versión optimizada para soluciones embebidas de OpenSSH y trata de adaptarse a sus entradas, salidas, líneas de comandos. No obstante, no son intercambiables, ya que mantienen diferencias importantes, principalmente en los archivos de configuración.

OpenSSH y DropBear utilizan el fichero *AuthorizedKeys* como fuente de claves públicas con las que identificar a los usuarios que desean ingresar en el sistema mediante clave público-privada. El fichero *AuthorizedKeys* es único para cada usuario del sistema Linux.

Cada línea del *AuthorizedKeys* contempla una clave pública a la que se permite el acceso bajo una serie de limitaciones configurables, que permiten estas funcionalidades:

1) impedir que se reserve terminal de sistema.
2) que se envíen sesiones X11.
3) limitar los puertos por proxificación directa a los que se pueden acceder.
4) la obligada ejecución de un comando.
5) la recepción del comando ejecutado para establecer la sesión SSH.

Una adecuada configuración del fichero a*uthorized_keys* permite modular la capacidad de conexión de un cliente SSH al servidor. Para completar la seguridad ofrecida se recurre al módulo *pam_limits* presente en los servidores Linux y que opera junto a SSH. *Pam_limits* permite, a través del archivo /etc/security/limits.conf, definir el número máximo de descriptores abiertos por usuario.

A continuación, se listan los principales escenarios posibles de ataque a los que se puede hacer frente simplemente con una configuración correcta del fichero a*uthorized_keys.* En los ejemplos incluidos a continuación, se considera que un dispositivo IoT ha sido *hackeado* y su llave comprometida. La correcta configuración del fichero a*uthorized_keys* permite mitigar con éxito el alcance de dicho ataque sin necesidad de modificar la implementación SSH:

1) El dispositivo se registra en el servidor IoT y trata de acceder a la línea de comandos.
    - El acceso a la línea de comandos es denegado por el servidor SSH, ya que la clave privada utilizada sólo puede emplearse para realizar proxificaciones directas o inversas. Específicamente, esto se consigue con la opción "*no-pty*" y "*no-user-rc*"

asignando un comando por defecto a la clave.

2) El dispositivo se registra en el servidor IoT y trata de acceder a otros servidores usando su clave por medio del "*agent forwarding*".
   - La opción de "*agent forwarding*" no se utiliza en IoTsafe y se desactiva en el servidor SSH. Si se requiere por algún otro motivo, se puede desactivar específicamente para cada clave pública usada por los dispositivos IoT desde el fichero *authorized_keys* con la opción "no-agent-forwarding"
3) El dispositivo se registra en el servidor IoT y trata de enviar peticiones X11 al servidor IoT.
   - La opción de "no-X11-forwarding" previene este ataque.
4) El dispositivo se registra en el servidor IoT y trata de realizar proxificaciones directas hacia puertos del servidor no autorizados o hacia otros puertos para que sean accesibles localmente desde el dispositivo. El dispositivo también puede reservar más proxificaciones reversas de las que debería, y tratar así de agotar los recursos del servidor:
   - OpenSSH dispone de la opción *permitopen*="host:port" y *permitlisten*="[host:]port" que limitan respectivamente ambos tipos de proxificaciones. En base a las configuraciones del dispositivo IoT, los parámetros *permitopen* y *permitlisten* se adecúan para que OpenSSH permita únicamente acceder y reservar los recursos permitidos. Estas opciones están disponibles únicamente en OpenSSH (a partir de la versión 7.8p1-1)
   - Dropbear no proporciona ningún mecanismo para limitar este ataque. No obstante, al funcionar bajo contextos de seguridad, *permitopen* es redundante. Un servidor IoT empleando el servidor de Dropbear, todavía sería vulnerable a un ataque de denegación de servicio por un dispositivo IoT que tratase de generar múltiples proxificaciones reversas. Para evitar esto, se recurre a una configuración de /etc/security/limits.conf en la que se limita a cada usuario específicamente el número de descriptores abiertos que pueden tener.
5) Múltiples dispositivos IoT tratan de autenticarse constantemente para causar denegación de servicio.
   - Si los dispositivos intentan abrir más de una conexión simultánea, se rechazará, al configurarse en "/etc/security/limits.conf" *maxsyslogins* para cada usuario.
6) A través de Fail2ban[132], se bloquean fácilmente todas las direcciones IP de donde vengan solicitudes de conexión al servidor (de cualquier tipo) tras un número de fallos determinado.
7) Un dispositivo comprometido se conecta con éxito al servidor IoT y se hace pasar por un dispositivo diferente IoT ante la plataforma.

[132] "Fail2ban - Daemon to ban hosts that cause multiple authentication errors" https://www.fail2ban.org/wiki/index.php/Main_Page

- La plataforma IoT puede poner a disposición de los dispositivos IoT puertos en los que ofrecer servicios. Estos puertos son accesibles para las aplicaciones de los dispositivos IoT mediante proxificaciones directas. Dado que la aplicación ha delegado toda seguridad en IoTsafe, la plataforma no dispone de una manera de detectar qué dispositivo está conectándose a su puerto: el dispositivo no va a hacer autenticación. Si se requiere controlar la identidad de los accesos a los Socket_A ofrecidos, se puede proceder de distintas maneras, siendo una de ellas la implementación de un *proxy middleware* web.
- Si la proxificación es reversa, el destino de la misma puede resolverse consultando los "Registro_acceso" y "Registro_*socket*" en la base de datos.
- Una identificación adecuada prevé el uso de identificadores únicos para los dispositivos IoT.

E1.2. Procedimientos de cierre de proxificación

El cierre de las proxificaciones se puede deber a dos motivos:

- La solicitud explícita de dicho cierre por parte del Agente.
- Inactividad.

En la Fig. C-5, se describe el proceso de cierre de proxificación siguiendo un diagrama de secuencia por una implementación basada en SSH. Dicho cierre se debe al primer caso analizado. La correspondencia de *software* con los elementos definidos en el diseño es la ya referida en esta sección, con la excepción del SSHD *connection monitor*, que hace referencia al monitor de accesos. El monitor de accesos tiene normalmente dos formas de funcionar, tal y como se explica en su definición. El procedimiento explicado en la Fig. C-5, detecta que el proceso padre que solicitó la proxificación se cierra por solicitud del Agente_B (el cliente SSH del dispositivo_B). Este hecho desencadena el proceso de clausura de dicha proxificación, que involucra borrar los

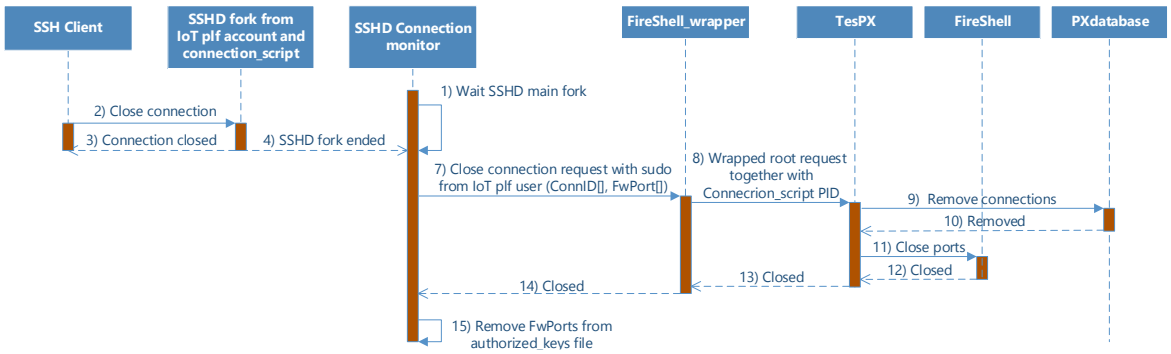


Fig. C-5: Ejemplo del diagrama de secuencia de una solicitud de cierre de proxificación en una implementación concreta basada en SSH.

registro_accesos pertinentes en la base de datos y quitar las reglas de marcado que habilitaban el contexto de seguridad pertinente que daba acceso a las proxificaciones reversas. Los registro_acceso de las plataformas IoT a las proxificaciones de los dispositivos suelen configurarse de ese modo. Por el contrario, un registro_acceso de un proceso diferente, que es temporal, se habilita de manera que, si existe un tiempo de inactividad superior a un umbral determinado, el valor tiempo de inactividad se deje de actualizar por el monitor de accesos (Fig. C-6) y este registro_acceso se cierre automáticamente. En el ejemplo de la Fig. C-7 ese límite es de 10 min.

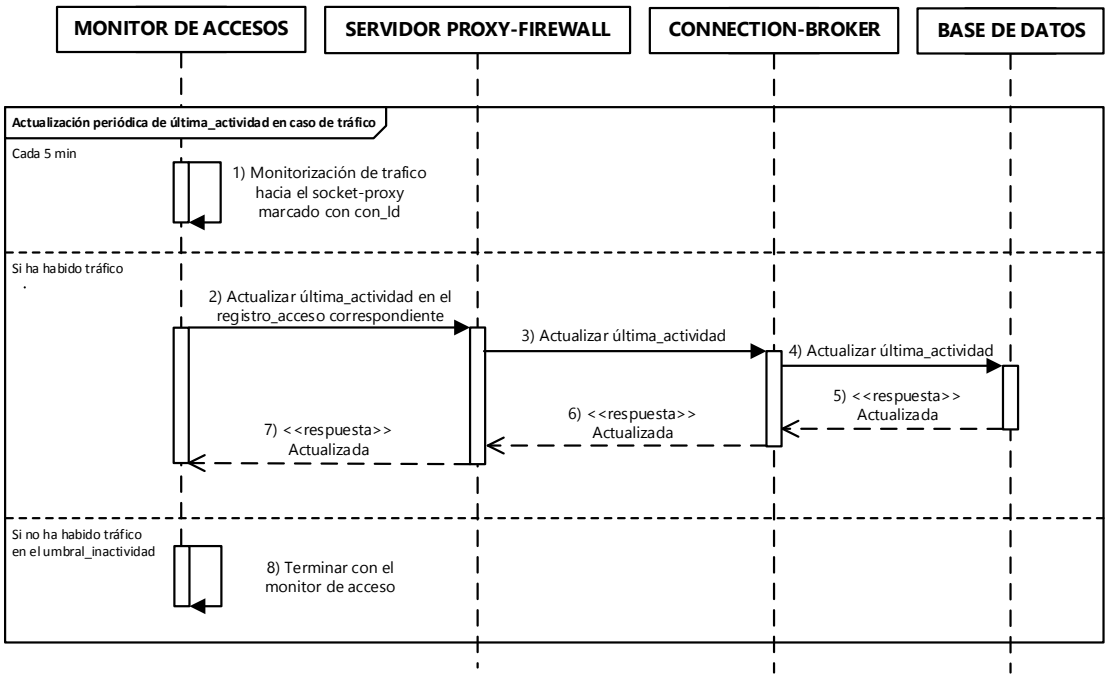


Fig. C-6: Diagrama de secuencia de monitorización del registro_acceso a un socket-proxy.

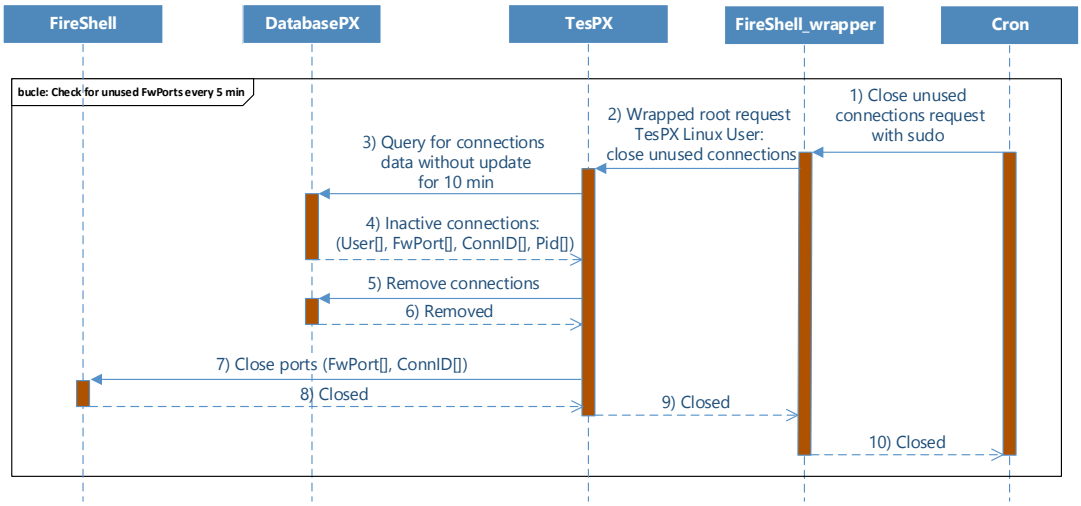


Fig. C-7: Ejemplo del diagrama de secuencia de una solicitud de cierre de proxificación por inactividad en una implementación concreta basada en SSH.

Por norma general, los registro_acceso vinculados a plataformas_IoT o a cuentas de usuario que van a proxificar nuevamente un *socket* protegido, son del primer tipo (no basados en inactividad sino en permanecer activos mientras dure el proceso que los creó, que es el Agente). Sin embargo, si este proceso muere por una interrupción puntual de la comunicación, para acelerar el restablecimiento de la conectividad, el sistema no cierra inmediatamente los registro_acceso vinculados a la proxificación perdida una vez muere el proceso. Espera el tiempo marcado para el umbral de actividad antes de cerrarlos definitivamente. De ese modo, si se reestablece la comunicación, las proxificaciones reciben los mismos *socket* asignados y los contextos de seguridad permanecen configurados, evitando tener que establecer nuevamente otros segmentos de la comunicación que estuviesen proxificando este *socket* protegido a otras máquinas remotas. En todo caso, cualquier de cierre de proxificaciones se regula de forma general según la Fig. C-8.

Cuando un *middleware* o una plataforma IoT disponen de un registro_acceso en vigor, con su contexto de seguridad correspondiente, la plataforma puede acceder en cualquier momento al *socket* protegido que habilita dicho contexto de seguridad. De ese modo, si la plataforma es multi-usuario y ofrece de forma directa los recursos de dicho *socket* protegido, deberá prestar especial atención a los procedimientos de sesión inherentes a su programación para evitar que nuevos usuarios puedan acceder a sesiones ajenas y conseguir el control sobre los datos de sesión (Fig. C-9).

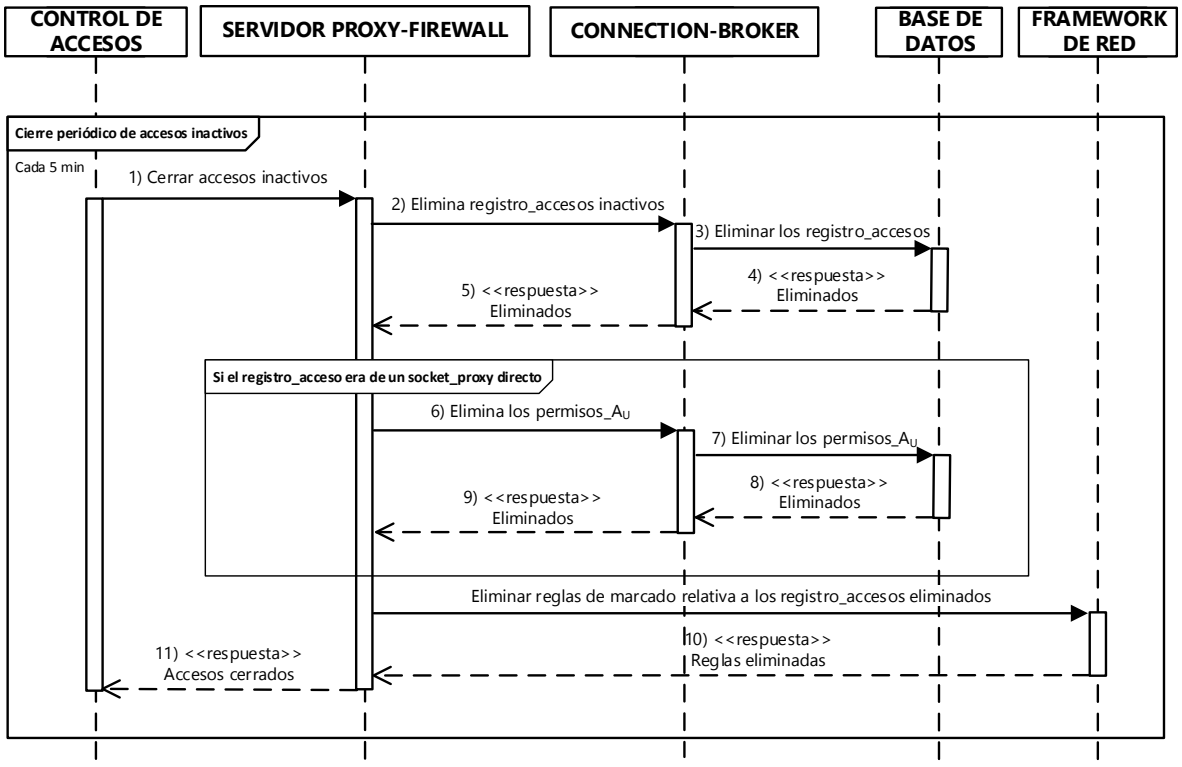


Fig. C-8: Diagrama de secuencia del cierre periódico de los registro_acceso sin actividad.

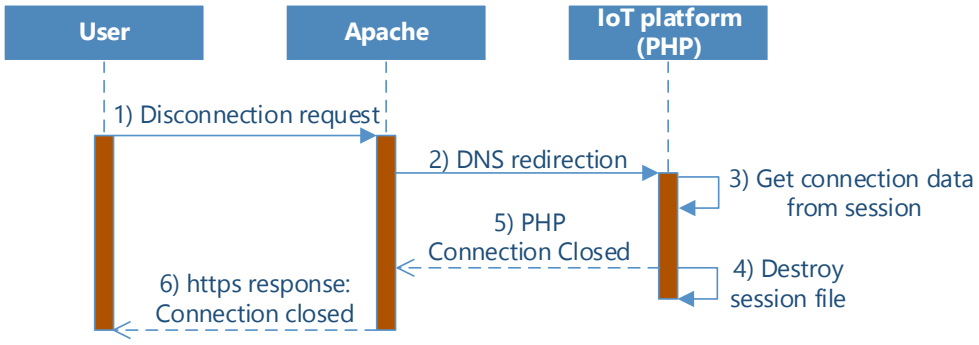


Fig. C-9: ***Ejemplo del diagrama de secuencia de una solicitud de cierre de sesión de acceso a una proxificación por un script del lado de servidor, en este caso, SSH.***

Ejemplo 2: Solicitud de un recurso de un socket protegido del dispositivo_A por un proceso_A

A continuación, se describe a modo de ejemplo un procedimiento por el que se concede acceso a un *socket* protegido del dispositivo_A al usuario del proceso_A para que éste pueda comunicarse con dicho *socket* protegido. Este procedimiento se ilustra con un diagrama de secuencia representado en la Fig. C-10. El registro_acceso y el contexto de seguridad se aplican en la cuenta de usuario local del dispositivo_A relativo al usuario_A. Los *socket* protegidos nunca pueden ser accesibles directamente desde máquinas remotas y sus accesos siempre son locales y hechos por usuarios dentro del contexto de seguridad. Esto es posible gracias a la separación de privilegios de sistemas POSIX que permite configurar los contextos de seguridad.

Para poder acceder al *socket* protegido (301), la cuenta usuario_A en la que se que ejecuta el proceso_A (302) que requiere el acceso debe disponer del correspondiente permiso de usuario en la base de datos (303) del *connection-broker* (304). El proceso_A realiza una solicitud (305) al servidor *proxy-firewall* (306) para realizar la resolución del *socket* protegido a partir del alias del *socket* y de su id_Disp*Socket* y de ese modo saber a qué *socket* local dirigir la petición del recurso. Dicha solicitud (305) se produce en este ejemplo mediante una llamada al sistema.

El origen de esta llamada no requiere del manejo de credenciales de usuario_A ante el servidor *proxy-firewall* en la propia solicitud ya que se aprovecha la separación de privilegios del sistema tipo POSIX, mediante la cual el servidor *proxy-firewall* puede identificar unívocamente la cuenta de procedencia de dicha llamada. Esta solicitud se valida (307) por el *connection-broker*, el cual inicialmente realiza una consulta (308) en la base de datos para verificar con su respuesta (309) si el usuario_A tiene permiso. Si el permiso es de tipo permiso_A_G, no se requiere de la concesión de un acceso y se procede con la respuesta (310) al servidor *proxy-firewall,* en la que se indica que se dispone de acceso y cuál es el *socket* protegido.

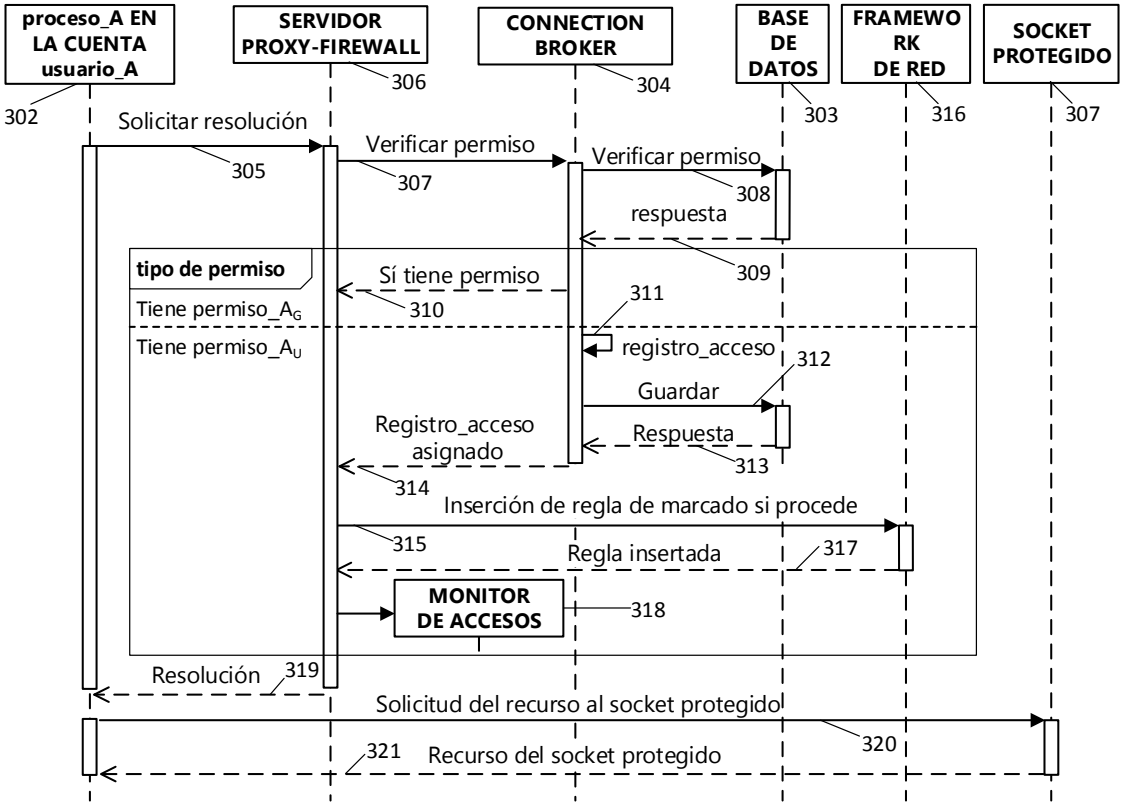


Fig. C-10: Ejemplo del diagrama de secuencia de una solicitud de un recurso a un socket protegido.

Sin embargo, si el permiso es de tipo permiso_A_U, el *connection-broker* debe conceder antes el acceso para el usuario_A al *socket* protegido. Para ello, el *connection-broker* genera un registro_acceso (311) y lo registra en la base de datos (312). Una vez recibida la confirmación (313) de la base de datos de que el registro se guardó correctamente y de que no existía uno previo, se comunica (314) al servidor *proxy-firewall* la necesidad de configurar el contexto de seguridad estricto para el usuario_A.

El servidor *proxy-firewall* utiliza la información del registro_acceso para definir la regla de marcado necesaria que conceda el acceso de las comunicaciones del usuario_A al *socket* protegido. Estas reglas de marcado se insertan (315) en el sistema operativo por medio del *framework* de red (316). Si hubiese un registro previo válido, no es necesario generar otro registro_ acceso para el usuario_A, porque ya dispone de acceso.

Tras recibir confirmación (317) de la inserción de las reglas (en caso de haber sido necesario), el servidor *proxy-firewall* instancia un monitor de acceso (318) para actualizar periódicamente el estatus de inactividad del acceso. En este ejemplo, la monitorización se hace a intervalos de 5 minutos. Este monitor de acceso, al igual que las reglas de marcado y que el registro_acceso, sólo se lleva a cabo si no existía otro registro_acceso válido anterior.

Una vez se confirma que el usuario_A dispone de un acceso al *socket* protegido, el servidor *proxy-firewall* notifica (310) (319) al proceso_A cuál es el *socket* protegido al que solicitaba acceder para que pueda enviar la solicitud (320) del recurso al *socket* protegido y recibir su respuesta (321).

Ejemplo 3: Acceso a un *socket-proxy* reverso por un proceso remoto a través de un proxy local del dispositivo_A que convierte peticiones del protocolo de aplicación https a peticiones http de forma transparente

En la Fig. C-11 se muestra un ejemplo de realización del dispositivo_A que permite poder acceder (401) (402) (403) a los *socket-proxies* (404) en el dispositivo_A (405) desde un proceso en un dispositivo diferente al dispositivo_A a través de conexiones proxificadas (406) en sesiones (407) establecidas con dispositivos remotos. El primer caso (401) es un ejemplo de este tipo de comunicación, en el que las aplicaciones que utilizan HTTPS acceden a recursos del *socket_B,* pero sin requerir que el proceso del *socket_B* de este dispositivo_B tenga que contar con soporte para HTTPS, ni esquema de autenticación. Este acceso se realiza a través de un *socket-proxy* reverso (en el dispositivo_A) de un *socket_B* de un dispositivo_B que tiene una sesión establecida con el dispositivo_A.

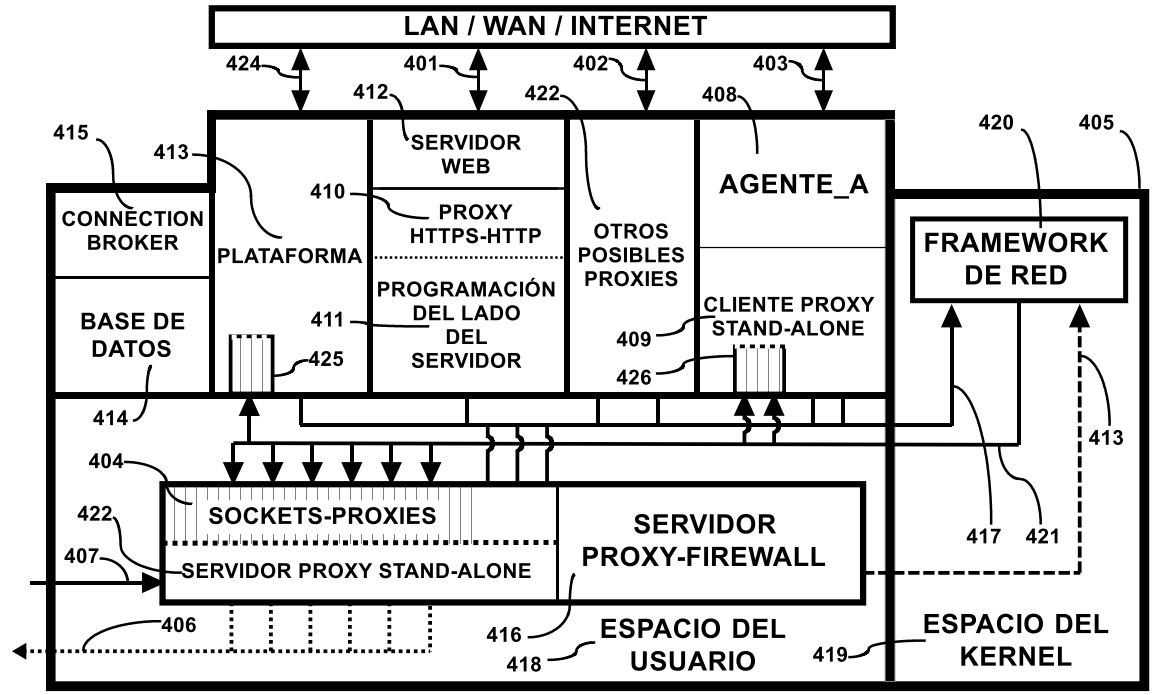


Fig. C-11: Esquema conceptual de IoTsafe en el Gateway o en el Server.

La comunicación entre la aplicación y el proceso del dispositivo_B es transparente y se efectúa a través de un *proxy* intermedio (410), que además convierte comunicaciones que utilizan HTTPS en comunicaciones HTTP. En primer lugar, la aplicación final solicita un recurso del dispositivo_B ofrecido por un *socket_B* proxificado en el dispositivo_A y dirige esta petición (401) hacia el *proxy* intermedio. Este *software*, que convierte la petición (501) de HTTPS a HTTP, puede estar implementado en multitud de lenguajes de programación del lado del servidor (411), como por ejemplo PHP, Node.js, Go, Python, etc.

En la Fig. C-12 se representa un diagrama conceptual en el que se produce un acceso a un recurso de un dispositivo IoT de forma transparente a través de IoTsafe, de manera que la aplicación final disponga de un acceso directo y transparente hasta el recurso ofrecido por el dispsitivo IoT. En el caso presentado, la aplicación final tramita una petición HTTPS (Fig. C-13) para un recurso que se ofrece en el dispositivo únicamente mediante HTTP. Un sencillo *middleware* de conversión de protocolos basta para habilitar dicha comunicación.

Para posibilitar dicho acceso al *socket-proxy* correspondiente, el *proxy* recibe la información del destino de la comunicación por medio de subdominios (502) y también recibe las credenciales que use la aplicación final para conseguir un acceso al *socket proxy* (estas últimas están incluidas en encabezados de la petición). El servidor web (412) debe estar configurado para redireccionar todas las peticiones en todos los subdominios posibles del *proxy* hacia la URL principal del *proxy* intermedio, incluyendo los subdominios como parámetros. De los encabezados, el *proxy* intermedio obtiene la credencial que la aplicación final adjunta y que permiten al *proxy* intermedio validar la petición y solicitar acceso al *socket-proxy*. Las credenciales van protegidas por HTTPS, al estar en los encabezados.

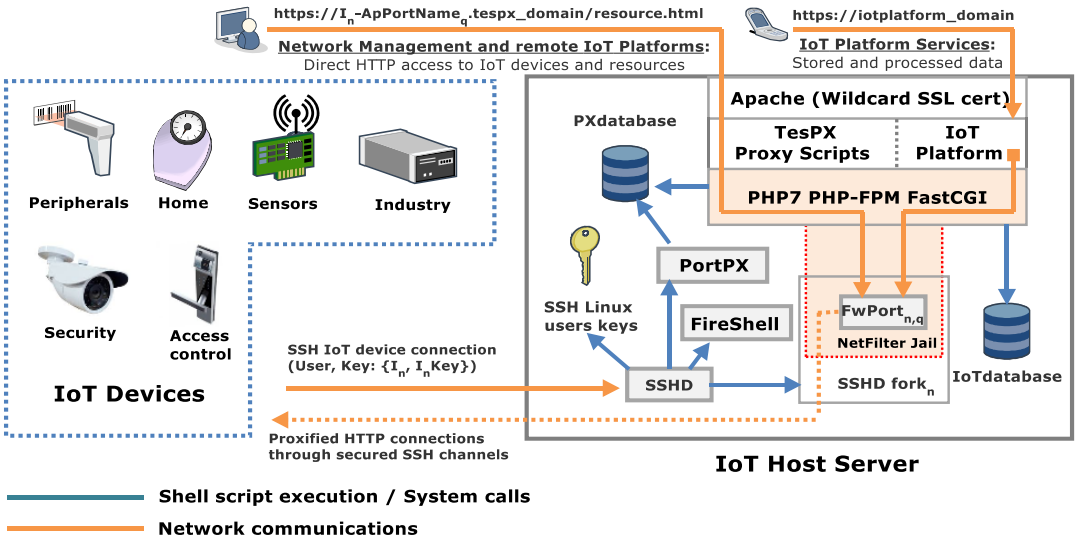


Fig. C-12: Diagrama conceptual de una implementación de IoTsafe empleando elementos software comunes.

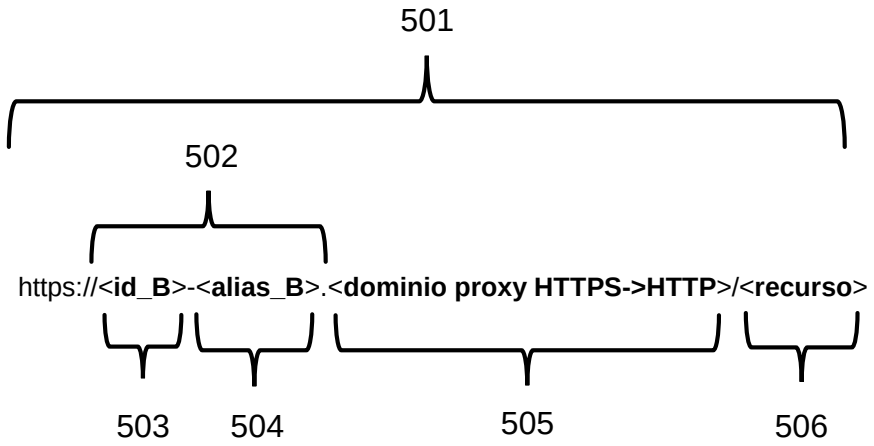


Fig. C-13: Ejemplo de una solicitud HTTPS para un acceso directo a un recurso IoT.

La petición que recibe el *proxy* intermedio cuenta con los siguientes elementos:

- El id_DispSocket, que en este caso corresponde con el id_B (503) del *socket_B* al que conduce el *socket-proxy* reverso, y que permite identificarlo junto al alias.
- El alias (504) del *socket_B* que permite identificar el *socket-proxy* junto al id_DispSocket.
- El dominio (505) del *proxy*.
- El recurso (506) que se solicita al *socket_B*.

La protección de las comunicaciones en cualquiera de los subdominios dinámicos que se requieran es posible con un único certificado *wilcard* SSL basado en certificados X.509, que habilita la protección de HTTPS para el dominio principal del *proxy* y de todos los subdominios posibles. Por otro lado, la comunicación extremo a extremo también es transparente e independiente de otras comunicaciones que pueda realizar la aplicación final con otros *socket-proxy,* al necesitar cada uno un subdominio diferente.

Las credenciales que van en los encabezados son utilizadas por el *proxy* intermedio para validar el acceso al *socket-proxy* para la cuenta de usuario, en la que se ejecuta el *proxy* intermedio según los permisos en la base de datos (414) para esa cuenta de usuario. Este acceso se puede solicitar directamente al *connection-broker* (415) a través del servidor *proxy-firewall* (416) o requerir una validación previa de la plataforma. Una vez realizada la conversión de HTTPS a HTTP y haberse habilitado el acceso al *socket-proxy* para la cuenta de usuario que ejecuta el *proxy* intermedio, éste suprime los encabezados con las credenciales, para hacer transparente el proceso de autenticación y envía la petición web HTTP (417) hacia el *socket-proxy* que corresponda.

Esta petición, al pasar del espacio de usuario (418) al espacio del *kernel* (419) es marcada por el *framework* de red (420), si el acceso se concedió correctamente y se insertó la regla de marcado correspondiente. Posteriormente, esta petición es enrutada y dirigida (421) hacia el *socket-proxy* y

será cursada o descartada según las reglas de filtrado, pudiendo únicamente ingresar de nuevo al espacio de usuario si posee un marcado válido. Tanto las reglas de marcado como de filtrado se introducen en el *framework* de red del sistema operativo por iniciativa del servidor *proxy-firewall* siguiendo las indicaciones descritas los ejemplos 1 y 2. Finalmente, la petición es recibida por el *socket-proxy* del servidor *proxy stand-alone* (422) y transmitida como comunicación proxificada hacia el dispositivo_B.

En la Fig. C-14 se describe mediante un diagrama de secuencia dicho procedimiento en una implementación que utilice SSH, Apache y un *connection-broker* en PHP (TesPX), tal y como se representa en la Fig. C-12. Este es un ejemplo completo en el que se considera que el *proxy* HTTPS<->HTTP no disponía de acceso por IoTsafe a dichos recursos. Además, este diagrama de secuencia incluye una salvedad particular: para mejorar el control del acceso a los recursos de los dispositivos IoT, se valida dicho acceso en la plataforma IoT (paso 3,4,5 y 6). Esta secuencia no es necesaria para el funcionamiento de IoTsafe, que únicamente regula los permisos y los accesos a usuarios de cuentas Linux. La plataforma utiliza siempre el mismo usuario Linux independientemente del usuario que haya utilizado la aplicación final para autenticarse en ella y solicitar el recurso del dispositivo.

Dado que el acceso al recurso debe poderse realizar sin una API, es decir, de forma directa, es necesario que se valide que dicha petición la realiza un usuario con una sesión válida y con unos permisos adecuados para ese recurso. De ese modo, en este caso, TesPX también juega el papel de *proxy* HTTPS <-> HTTP transparente, utilizando un *proxy* Guzzle, que permite realizar este tipo de procedimientos. En particular, los datos que permitan validar el acceso a dicho recurso protegido deben figurar en el encabezado de la petición directa HTTPS. Esto permite a Guzzle extraer de forma transparente dicha información y traducir la petición a otra diferente HTTP que ya no la tenga en los encabezados.

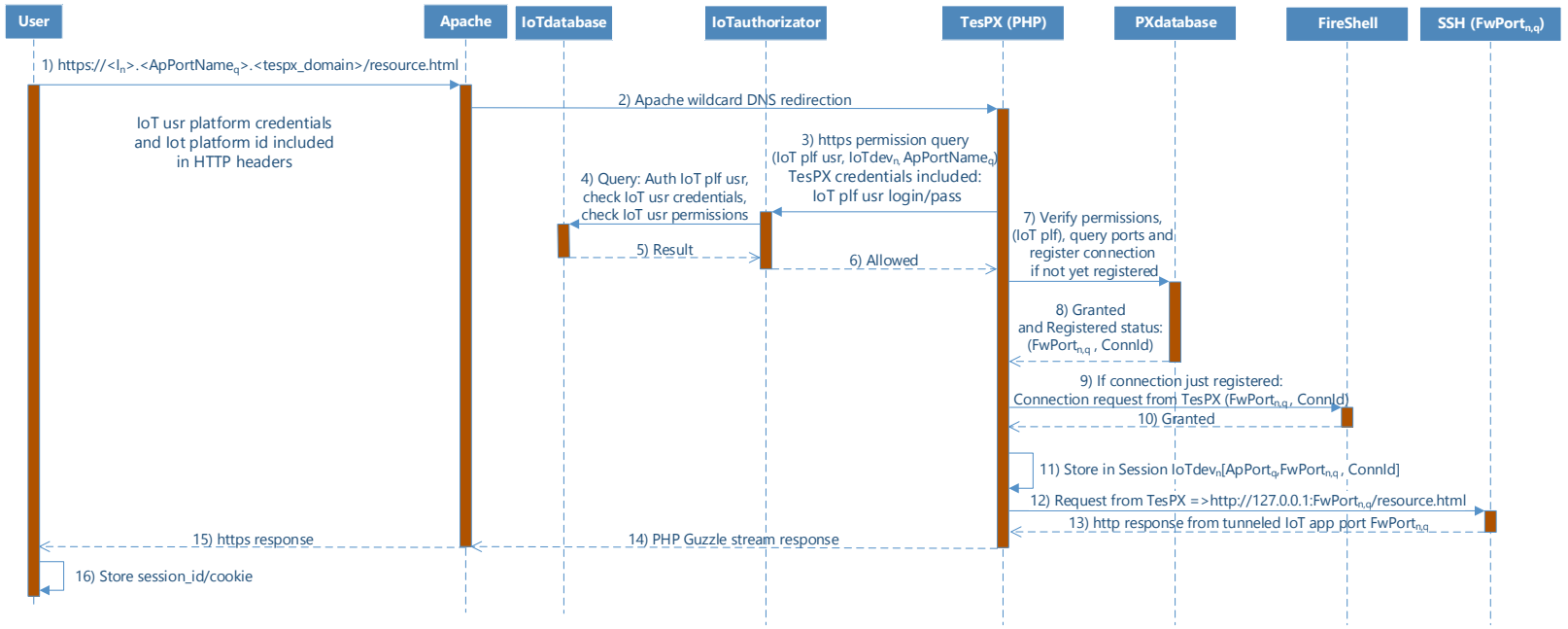


Fig. C-14: Diagrama de secuencia de una implementación de IoTsafe en la que se solicita un recurso de un dispositivo IoT de forma directa a través de una llamada HTTPS.

Por otro lado, en la Fig. C-15 se representa un diagrama de secuencia más sencillo en el que el *proxy* HTTPS<->HTTP incluido en TesPX ya dispone de un acceso concedido por IoTsafe, y además, TesPX mantiene una sesión válida con la aplicación final, evitando tener que enviar, por cada petición, datos de autenticación que permitan validar a TesPX el acceso al recurso solicitado en la plataforma IoT. Esto acelera notablemente la comunicación directa con recursos de dispositivos IoT a través de proxies intermedios, pero sin APIs en origen ni destino.

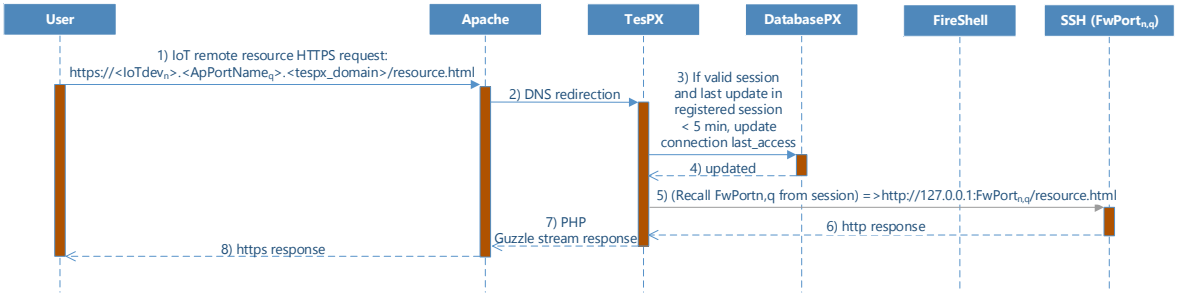


Fig. C-15: Diagrama de secuencia de una implementación de IoTsafe en la que se solicita un recurso de un dispositivo IoT de forma directa a través de una llamada HTTPS en el que ya existía un contexto de seguridad establecido previamente.

E3.1. Funcionamiento de los contextos de seguridad en el Ejemplo 3

En los apartados anteriores se ha explicado el proceso de comunicación seguido en la implementación particular de IoTsafe de la Fig. C-12. En ese ejemplo, una aplicación de usuario final utilizando HTTPS como protocolo de comunicación, requiere acceder a un recurso ofrecido por un dispositivo IoT, pero de forma transparente. En ese ejemplo, TesPX hace las funciones de *connection-broker* y de *proxy* HTTPS<->HTTP.

En dicho ejemplo, se explica que IoTsafe solamente se ocupa de los permisos a nivel de cuenta de usuario Linux. Esto quiere decir que, si la plataforma IoT que utiliza el *proxy* HTTPS<->HTTP no lo configura adecuadamente para regular el acceso a las proxificaciones en función de los permisos de los usuarios registrados en la plataforma, cualquier usuario de la plataforma podría acceder a cualquier *socket* protegido al que tenga acceso la cuenta de usuario Linux que ejecuta esa plataforma. Es por ello que en la Fig. C-14 se contemplan los pasos 3, 4, 5 y 6; en los que se pide autorización inicialmente a la plataforma en base a los datos de sesión que se tiene con ella, y en función de los datos de acceso del usuario de la plataforma en los encabezados (en caso en que la sesión no se propague a subdominios).

La petición remota se tramita entonces (desde el punto de vista del contexto de seguridad) según la Fig. C-16. En esa figura, se explica paso a paso (del 1 al 9) el procedimiento que sigue dicha petición hasta ofrecer la respuesta. Según la definición de contexto de seguridad y *socket*

protegido, existen en juego dos tipos de reglas: Reglas generales (Default rules) y Reglas prioritarias o específicas (Specific rules).

Cuando llega la petición HTTPS a TesPX, éste la procesa, verifica el usuario de la plataforma que la solicita (vía sesión o encabezados) y la convierte en petición HTTP, orientándola hacia el *socket* protegido correspondiente. Dicho *socket* protegido se identifica a partir de la información en el subdominio (Id de dispositivo y alias de servicio).

TesPX, aunque en el ejemplo sea un *script* PHP, se ejecuta bajo una cuenta de usuario Linux determinada sin privilegios. Según las definiciones, esta cuenta la podemos nombrar como cuenta Usuario_A. Todas las cuentas Linux del servidor se ven afectadas por unas reglas de marcado y filtrado generales que limitan el uso que se puede dar a cierto rango de direcciones IP privadas (pool de direcciones).

Dado que todas las proxificaciones se establecen en ese rango de direcciones, todas ellas quedan afectadas por las reglas generales:

- **1ª Regla general (de tipo filtrado):** Descartar todo el tráfico con origen o destino una dirección reservada procedente de cualquier IP que no pertenezca al pool de direcciones reservadas.
- **2ª Regla general (de tipo marcado):** Todo tráfico procedente de una dirección IP local protegida, debe marcar el campo ToS a cero.
- **3ª Regla general (de tipo filtrado):** Descartar todo tráfico generado con origen o destino una dirección de red reservada sin marcado en el campo ToS.

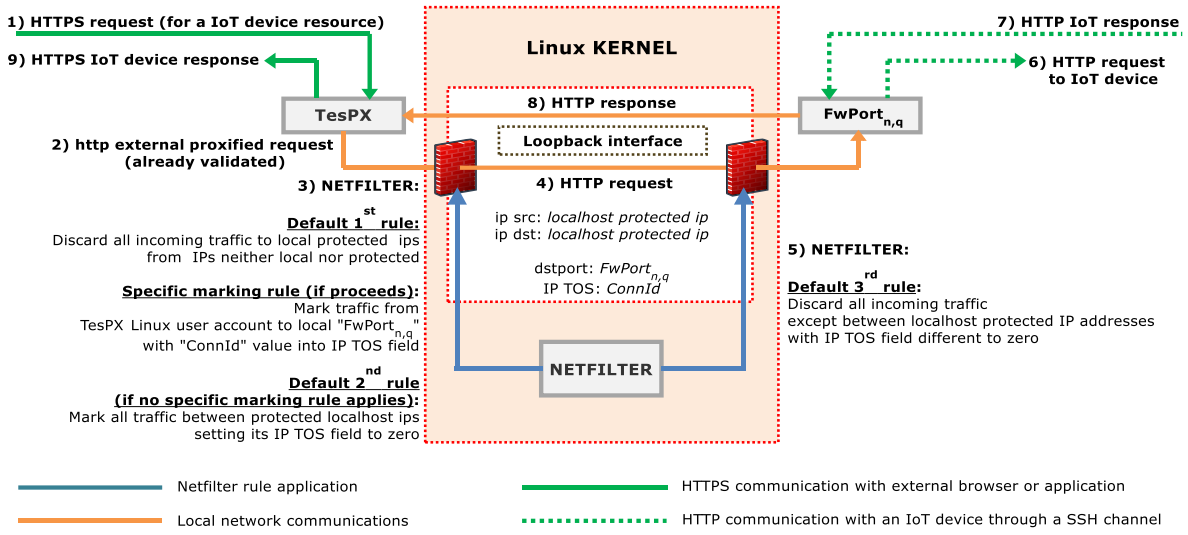


Fig. C-16: Ejemplo de la aplicación de las reglas de marcado y filtrado por Netfilter configurando un contexto de seguridad que permita un acceso transparente a un socket protegido de un dispositivo IoT desde una aplicación final externa.

El marcaje en el campo ToS se puede producir de forma general desde un programa o por reglas del framework de red (Linux Netfilter). En todo caso, el marcaje desde Netfilter se produce siempre después de que el paquete haya abandonado el espacio usuario y, por tanto, cualquier marcaje que de cualquier programa efectue sobre sus paquetes se sobre escribe con la segunda regla general de Netfilter siempre. Por consiguiente, desde el espacio usuario, sin acceso a Netfilter, es imposible realizar marcaje de paquetes en el campo ToS en el tráfico que involucre alguna comunicación entre direcciones protegidas, y cualquier tráfico entre las direcciones reservadas y las no reservadas es descartado siempre. Esto evita que desde el espacio usuario se pueda interferir con el funcionamiento de IoTsafe.

La habilitación de un acceso mediante un contexto de seguridad se reduce a incluir una nueva regla de marcado prioritaria sobre la general. En Netfilter, las reglas se aplican a los paquetes de forma lineal, de manera que cuando una regla de la tabla de filtrado es válida, las demás reglas se ignoran. El orden de aplicación de las reglas es de tipo LIFO, es decir, se evalúan las últimas en insertarse, y las primeras en introducirse en las tablas (las de por defecto) son las últimas en evaluarse. De ese modo, si existe una regla de marcado de paquetes que sea aplicable a algún paquete, este se marca con esa regla antes de llegar a la regla de marcado con cero, y una vez marcado con un valor no nulo, Netfilter deja de evaluar reglas de marcado para ese paquete.

Netfilter, por otro lado, es un *firewall stateful*, lo cual indica que es capaz de permitir las respuestas de un flujo de comunicación permitido. De este modo, si se autoriza un flujo de datos de un *socket* particular a otro, las respuestas de este último podrán cursarse sin problemas hacia el primero (al no ser que se especificase lo contrario con reglas complementarias).

En la Fig. C-12, el proceso de comunicación queda del siguiente modo:

1) Se recibe una solicitud HTTPS como la de la
2) Fig. ***C-13***, en la que se solicita un recurso de un dispositivo IoT concreto. Dicha solicitud incluye en los encabezados los datos de autenticación con la plataforma encargada de gestionar ese dispositivo, o la petición cuenta ya con una sesión válida con el *proxy* TesPX.
3) Tras haberse validado la solicitud HTTPS, el *proxy* la traduce a petición HTTP y le asigna el destino correspondiente al *socket* protegido con la proxificación reversa que conduzca al *socket* del dispositivo IoT que ofrezca el recurso que se necesita. Dicho *socket* protegido en la figura es $FwPort_{n,q}$, el cual se averigua tras resolver el ID del dispositivo y el alias del *socket_B* en la base de datos del *connection-broker* (Pxdatabase del TesPX).
4) La petición entra en Netfilter y se evalúa la regla general 1: no se produce el descarte, ya que la comunicación se produce entre direcciones protegidas a las que pertenece el *socket* $FwPort_{n,q}$ y la dirección local que usa TesPX para estas comunicaciones.

5) Regla prioritaria de marcado: Dado que TesPX tiene permisos para acceder a $FwPort_{n,q}$, al tramitar la resolución del mismo se insertó en Netfilter una regla de marcado prioritaria, que determina que todo tráfico del usuario Linux que ejecuta TesPX quede marcado con el valor ConnID, si el destino de dicha comunicación es $FwPort_{n,q}$,(*socket* en una IP local protegida). Al quedar marcado con esta regla prioritaria, la segunda regla general se salta (y cualquier otra regla prioritaria también).
6) Regla general 3 de filtrado: La comunicación, al tener marcaje en el campo ToS distinto de cero, no se descarta y se permite su acceso a $FwPort_{n,q.}$
7) La petición se comunica de forma transparente hasta el dispositivo IoT donde reside la aplicación con el *socket* que ofrece el recurso.
8) La aplicación responde y dicha respuesta vuelve.
9) La respuesta llega a TesPX por ser Netfilter un *firewall stateful*.
10) TesPX opera como *proxy stateful* y traduce la respuesta de HTTP a HTTPS, enviándola a la aplicación final.

De este modo, la comunicación se produce de forma transparente para la aplicación final. El procedimiento de marcado y filtrado (establecimiento de contextos de seguridad) se aplica de igual modo a cualquier otro proceso que ejecute cualquier usuario Linux del *Gateway*/server.

E3.2. Protección del Gateway/Servidor IoTsafe frente a ataques derivados de un dispositivo hackeado

IoTsafe se apoya de funcionalidades Linux para mantener un esquema simple que le permita ofrecer todas las funcionalidades de seguridad requeridas para los proyectos IoT. Estas funcionalidades se detallan en la Fig. C-17 con las letras a, b, c y d:

a) Protección de los dispositivos IoT: Las comunicaciones de los dispositivos IoT deben hacerse a través de la sesión cifrada que IoTsafe mantiene con el servidor. El resto de comunicaciones (entrantes y salientes) son descartadas.
b) Las comunicaciones entre los dispositivos IoT y los servidores IoT se llevan a cabo mediante sesiones SSH.
c) Todas las comunicaciones locales llevadas a cabo en el servidor IoT son gestionadas y protegidas por Netfilter. Sus reglas relativas al tráfico IoT se definen por el *software* IoTsafe y los contextos de seguridad.
d) Las cuentas de Linux están sometidas a una estrecha vigilancia por parte del *Computer Security Model* (p.ej: SELinux o AppArmor). Estos "*security models"* también interactúan con Netfilter proporcionando una protección complementaria. Principalmente se filtran los log de Netfilter para llevar a cabo contramedidas en caso de detectar comportamientos

anómalos de alguna cuenta IoT: no respeto de las reglas definidas por Netfilter (y éste tiene que descartar), solicitud de proxificaciones no concedidas o envío de tráfico no autorizado.

Es importante tener en cuenta que SELinux y AppArmor se consideran herramientas estándar ya habilitadas de forma predeterminada en ciertas implementaciones Linux *server* (CentOS / Redhat y Ubuntu). Por tanto, los administradores de sistemas Linux o ingenieros de seguridad pueden estar ya familiarizados con las herramientas. De este modo, se puede implementar fácilmente una configuración de seguridad adecuada requerida por IoTsafe para funcionar sin necesidad de contar con personal de cualificación más específica.

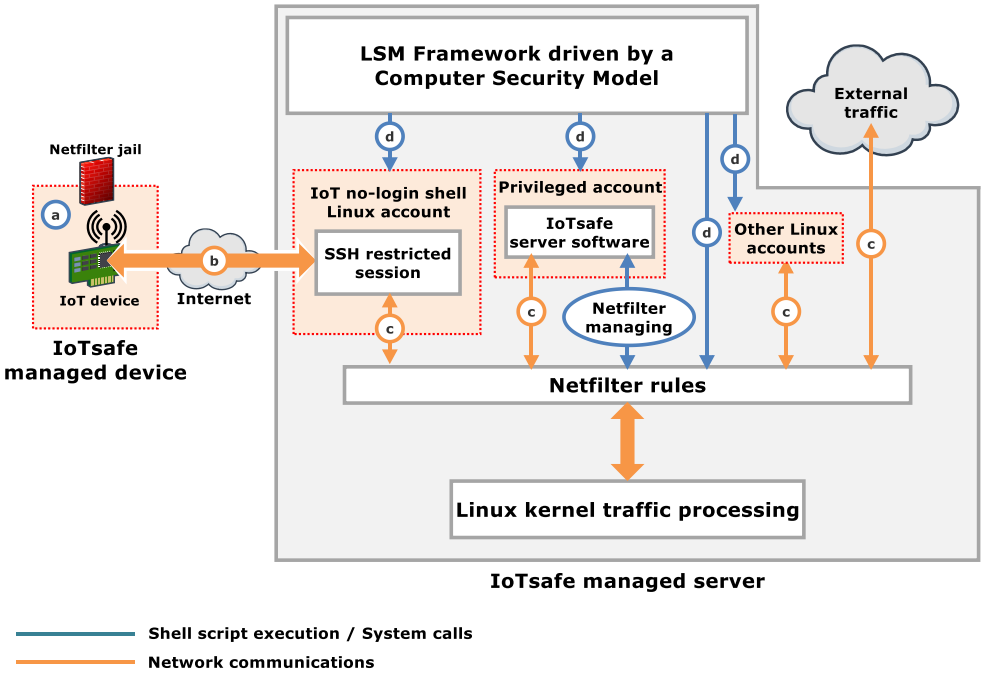


Fig. C-17: Example of NetFilter rules usage in an external transparent access to an IoT device resource.

E3.3. Diagrama de clases de un sistema IoTsafe para el Ejemplo 3

A continuación, en la Fig. C-18 se representa un diagrama de clases que incluye los elementos necesarios para elaborar un *Gateway*/Server y los elementos que componen un dispositivo IoT para operar según el comportamiento descrito en el ejemplo 3. El diagrama de clases no hace referencia a ninguna implementación particular y emplea los conceptos originales descritos en la lista de definiciones del presente documento.

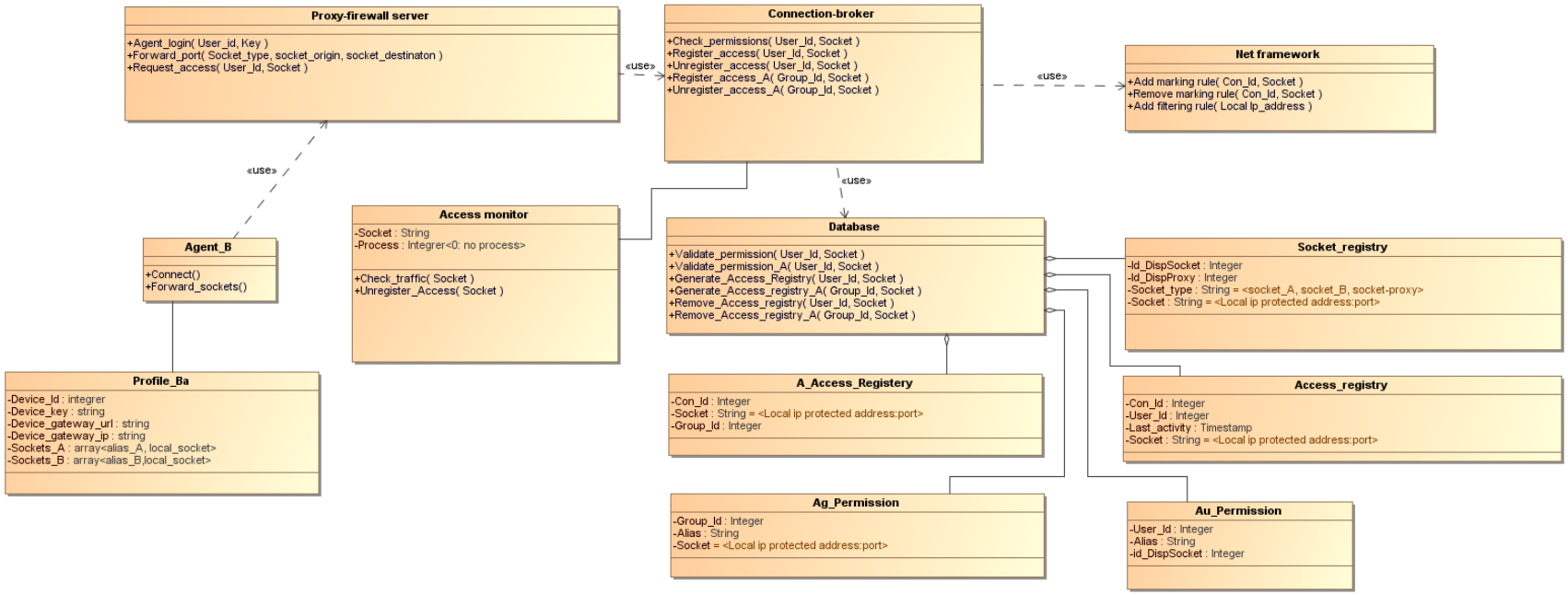


Fig. C-18: Diagrama de clases para una implementación básica de un Gateway/Server de IoTsafe. En este diseño no se incluyen clases extra requeridas para la comunicación entre Gateway ⇔ Server o entre Server ⇔ Server.

E3.4. Separación de privilegios aplicada en combinación con la plataforma IoT

En aras de simplificar el diseño de IoTsafe, la separación de privilegios se realiza aprovechando las ventajas inherentes al Shell de Linux:

- El fichero **/etc/sudoers** permite definir permisos de ejecución con privilegios elevados: permitir a usuarios o grupos de usuarios ejecutar comandos específicos.
- Para permitir a una plataforma IoT acceder a los puertos protegidos que correspondan se deben solicitar previamente los establecimientos de los contextos de seguridad mediante el *script* **Connection_*script*.sh**. Este *script* se encuentra habilitado en /etc/sudoers para poderse ejecutar con altos privilegios por el que lo requiera.
- Connection_*script*.sh valida el usuario que realiza la llamada al sistema para la ejecución y comprueba por medio de TesPX.php de que la plataforma IoT disponga de los permisos necesarios para acceder a los *socket* protegidos requeridos.
- TesPX habilita las reglas de marcado necesarias (una por cada *socket* protegido) para habilitar al usuario IoT_usr_plf a acceder a los *sockets* protegidos. También registra en la base de datos un registro_acceso por cada *socket* protegido habilitado.

La Fig. C-19 analiza la separación de privilegios en la implementación específica analizada, para peticiones externas. En esta implementación, TesPX.php (*connection broker* y y HTTPS-HTTP) recibe la petición HTTPS. De la petición se infiere (por su dominio) la plataforma IoT encargada de validar dicha petición. Del subdominio se extrae el Id del dispositivo y el alias del *socket_B*. En los encabezados van insertados el usuario y el *password*. Esta información se transmite por una comunicación HTTPS entre TesPX.php y el browser que realiza la solicitud. Una vez la plataforma IoT valida las credenciales, y que el usuario dispone de acceso al recurso del dispositivo, TesPX también valida que la plataforma tenga acceso al dispositivo. Si ambas validaciones son correctas, TesPX introduce en la base de datos el registro_acceso correspondiente (si no se disponía de él) y ejecuta *fireshell* con sus permisos elevados para introducir el marcado de los paquetes (si no existía el registro_acceso). Esta estrategia de dos niveles sirve para favorecer la diversificación de servicios, permitiendo a múltiples tipos de plataformas IoT coexistir sin limitar la libertad de su diseño.

E3.5. Autenticación transparente de los dispositivos IoT

La autenticación transparente de los dispositivos IoT al acceder a puertos de la plataforma puede conseguirse de múltiples maneras. En la Fig. C-20 se detalla conceptualmente un

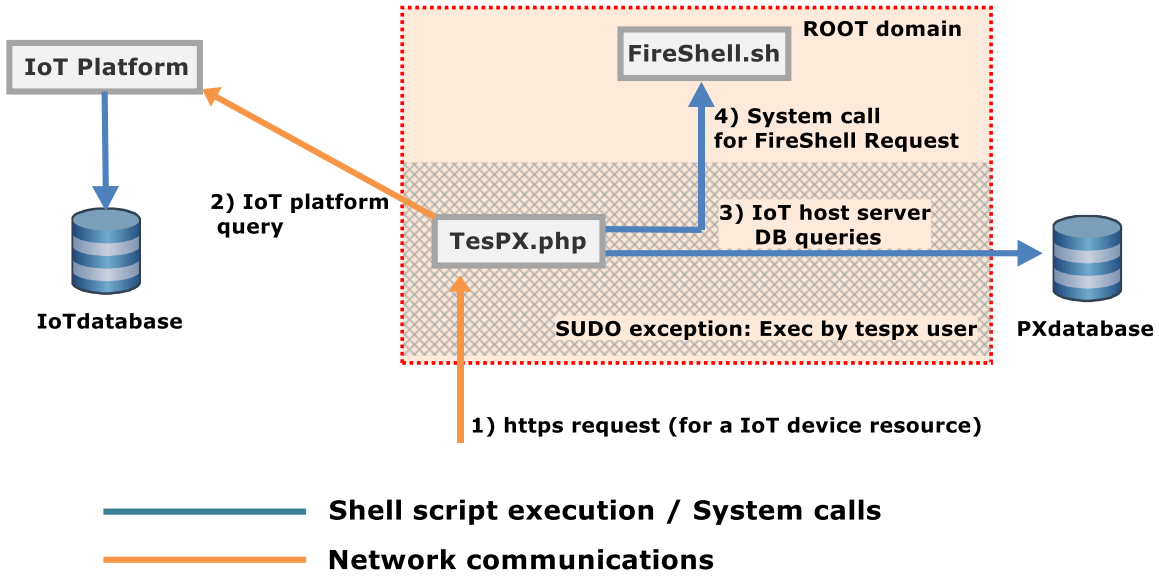


Fig. C-19: ***Separación de privilegios en caso de petición de un recurso de un dispositivo IoT desde una fuente externa.***

procedimiento a seguir si el Socket_A, al que los dispositivos IoT desean acceder, es web y utiliza el protocolo HTTP, aunque esta solución sería posible extenderla a otros protocolos de manera similar.

Durante la inicialización de las configuraciones, ServerPX lee el Profile_A (1) y comunica a la plataforma IoT (2) que debe cambiar el Socket_A por el Socket_A_M. Posteriormente, ingresa en la base de datos el Registro_acceso (para el usuario Socket_MDW al *socket_*A_M) y el Registro_acceso_A (Grupo del usuario_A para acceder al *socket_A*) (3). ServerPX también ingresa las reglas de marcado correspondientes en NetFilter (4). Posteriormente, configura PHP *proxy-middleware* para que redirija de forma transparente las comunicacones Socket_A al Socket_A_M (5). Este *middleware-proxy* se ejecuta en la cuenta de usuario Socket_MDW funcionando como *proxy* transparente basado en Guzzly. Este *proxy* PHP tiene la función de obtener el Id del dispositivo que se comunica con el *socket_A*. Para ello, a partir del *socket* origen de comunicaciones de la petición que llega al *socket_A*, identifica el usuario origen de la misma. Este usuario corresponde a un único dispositivo IoT.

La información del Id del dispositivo IoT se inyecta en un *header* de la petición HTTP cursada de modo que, si la plataforma IoT destino requiere averiguar el dispositivo origen de la petición, pueda obtenerlo a partir de los encabezados HTTP mediante los pasos (6) (7) (8) (9).

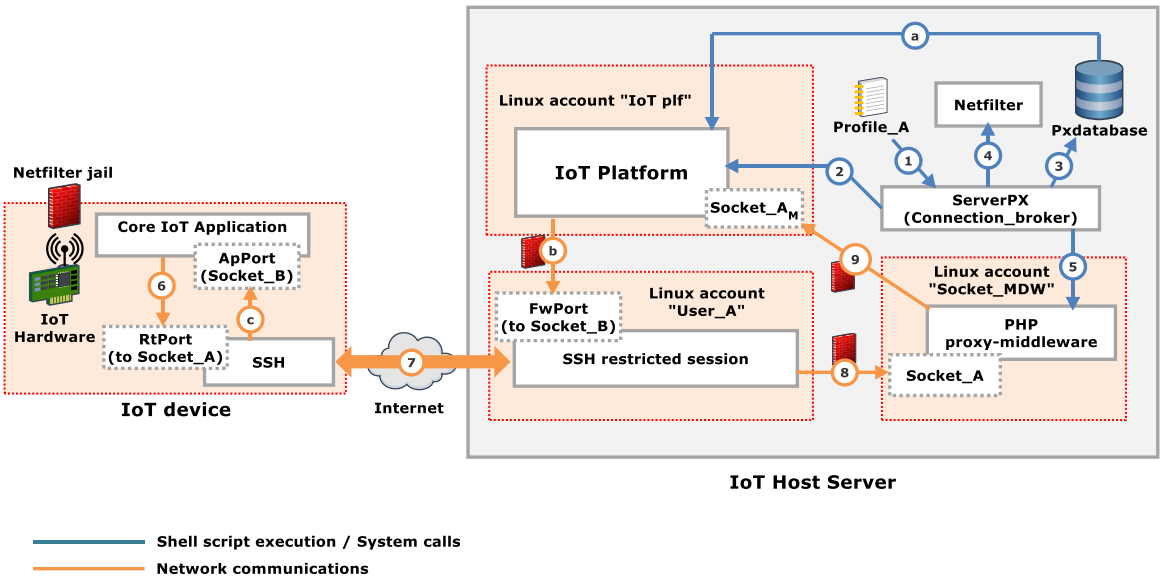


Fig. C-20: ***Diagrama conceptual representando una autenticación transparente de los dispositivos IoT mediante un framework ligero basado en encabezados de protocolo de aplicación.***

Por otro lado, los accesos a las proxificaciones reversas funcionan como se definieron inicialmente: la plataforma IoT solicita acceso y resuelve el *socket* protegido a partir del id del dispositivo IoT y el alias del *socket* (a). Una vez resuelto, accede al *socket* protegido local (b) que le comunica con el Socket_B destino en el dispositivo IoT (c).

Ejemplo 4: Acceso a un *socket-proxy* por un proceso remoto a través de un proxy intermedio del dispositivo_A en el que el proxy convierte las peticiones entrantes de la versión segura de un protocolo de aplicación a la versión insegura del mismo

En este ejemplo se generaliza el ejemplo anterior. Siguiendo el mismo esquema, el ejemplo se puede aplicar a otros protocolos del nivel de aplicación utilizados por aplicaciones de dispositivos_B que requieren de seguridad pero que no la implementan. Algunos ejemplos de estos protocolos pueden ser HTTP/2, MQTT y COAP.

Todos ellos tienen también una versión segura de sí mismo y con un *proxy* que convierta de forma transparente peticiones seguras a peticiones inseguras se hace factible que los dispositivos_B utilicen la versión insegura del protocolo junto con el sistema expuesto en IoTsafe para conseguir una comunicación segura extremo a extremo.

Ejemplo 5: Acceso a *socket-proxies* por una plataforma que recopila información de ellos, la procesa, almacena y luego ofrece a las aplicaciones finales en forma de recursos propios

La comunicación (424) representa una modalidad de operación muy común en ecosistemas IoT. En este caso la aplicación final no requiere un acceso directo a los recursos de los *sockets* proxificados, sino a la información y servicios que la plataforma elabora a partir de esos recursos que obtiene de forma autónoma. La plataforma, para ello establece comunicaciones periódicas con los *socket* protegidos a los que tiene permiso para acceder, y de los que recopila información con la que luego elabora sus recursos y servicios propios, los cuales podría seguir ofreciendo aun cuando todas las sesiones (407) estuviesen caídas temporalmente.

El modo de operar del sistema propuesto por IoTsafe en este caso es análogo al del ejemplo 2. En particular, el proceso_A (302) es aquel que ejecuta la plataforma para comunicarse con el *socket* local que requiera en cada momento. La plataforma debe de disponer para ello de una cuenta de usuario del dispositivo_A con permisos_A_U suficientes para poder acceder a todos los *socket-proxy* relativos a los dispositivos configurados en la plataforma, y de los que obtiene los recursos periódicamente. Por otro lado, la plataforma también puede ofrecer información necesaria a otros dispositivos por medio de *socket* seguros de tipo *socket_A* (425).

En la Fig. C-21 se muestra de forma detallada los pasos 7,8,9 y 10 de la Fig. C-14 y se define la regla de marcado que, junto con las de filtrado ya existentes, configura el contexto de seguridad.

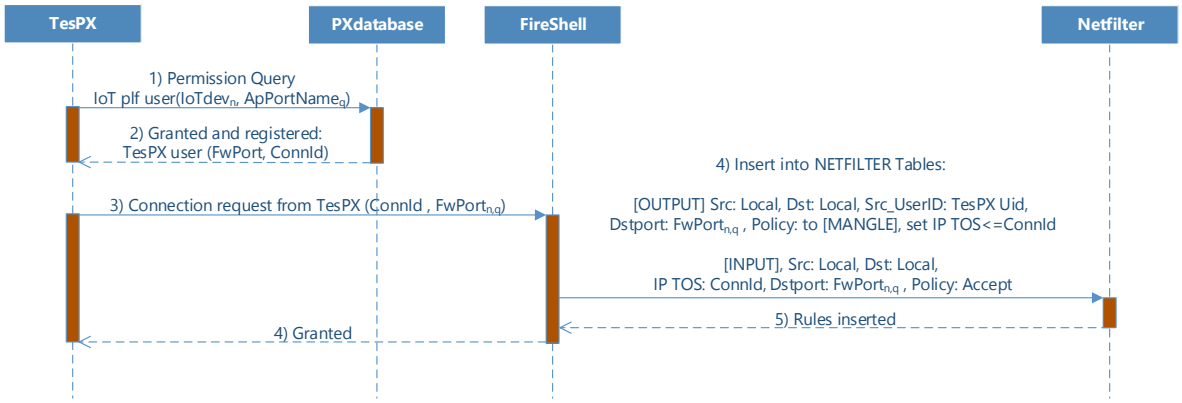


Fig. C-21: ***Diagrama de secuencia de una implementación de IoTsafe en la que se solicita un registro_acceso a un socket protegido por iniciativa de un programa en la cuenta de usuario IoTdevn (TesPX: Connection broker). Este diagrama detalla los pasos 7,8,9 y 10 de la Fig. C-14.***

Ejemplo 6: Solicitud de acceso a sockets protegidos de un dispositivo_AF por medio de proxificaciones directas y a instancia de un proceso_A

Este ejemplo (Fig. C-22) ilustra cómo el dispositivo_A, a instancias del proceso_A (601), solicita por medio del servidor *proxy-firewall* (602) y del agente_A (603) una proxificación directa (604), (426) de unos *socket* protegidos de un dispositivo_AF (605). Esto permite que el proceso_A pueda acceder de forma segura a recursos de *socket* protegidos en ese dispositivo remoto. Para ello, el proceso_A solicita (606) al servidor *socket-proxy* una resolución de *socket* remotos, suministrando el alias, el id_Disp*Socket* y el id_Disp*Proxy* de cada *socket* protegido al que se desea acceder. En este caso, el id_Disp*Socket* es distinto al id_Disp*Proxy* y, por ese motivo, el servidor *proxy-firewall* determina que se requieren proxificar de forma directa esos proxies desde otros dispositivos_AF. En el ejemplo, todos los *socket* a proxificar provienen del mismo dispositivo, pero no necesariamente tiene por qué ser así en otros casos.

La solicitud de proxificación directa realizada por el proceso_A al servidor *proxy-firewall* se realiza mediante una llamada al sistema que instancia el servidor *proxy-firewall* como proceso de ese entorno de usuario. Para proceder con la proxificación directa, el servidor *proxy-firewall* inicialmente solicita (607) al *connection-broker* (608) que reserve y registre (609) en la base de datos (610), unos *socket* locales libres que pertenezcan al pool de direcciones protegido para asignar los *socket-proxy* directos. La reserva de estos *socket* implica además la inserción de un registro_*socket* de tipo *socket-proxy*, y un registro_acceso si dispone de permisos_A_U para cada uno por cada *socket* reservado.

Una vez se confirma la reserva (611) de esos *socket* locales, el servidor *proxy-firewall* recibe en la respuesta (612) del *connection-broker* los *sockets* reservados y solicita (613) al agente_A el establecimiento (614) de una sesión con el dispositivo_AF para resolver los *socket* protegidos que se requiere proxificar. El agente_A utilizará un perfil_UA_{AF} relativo al usuario_A que se autentica en la cuenta usuario_AF del dispositivo_AF y que es suministrado por el servidor *socket-proxy*. Una vez se establece la sesión usando dichas credenciales, el dispositivo_AF realiza las acciones pertinentes (615), que serán descritas en el ejemplo 7, para procesar la solicitud de proxificación recibida. Una vez el dispositivo_AF responde (616) al agente_A con los *sockets* protegidos resueltos, el agente_A notifica (617) al servidor *proxy-firewall* y procede a establecer las proxificaciones directas (604).

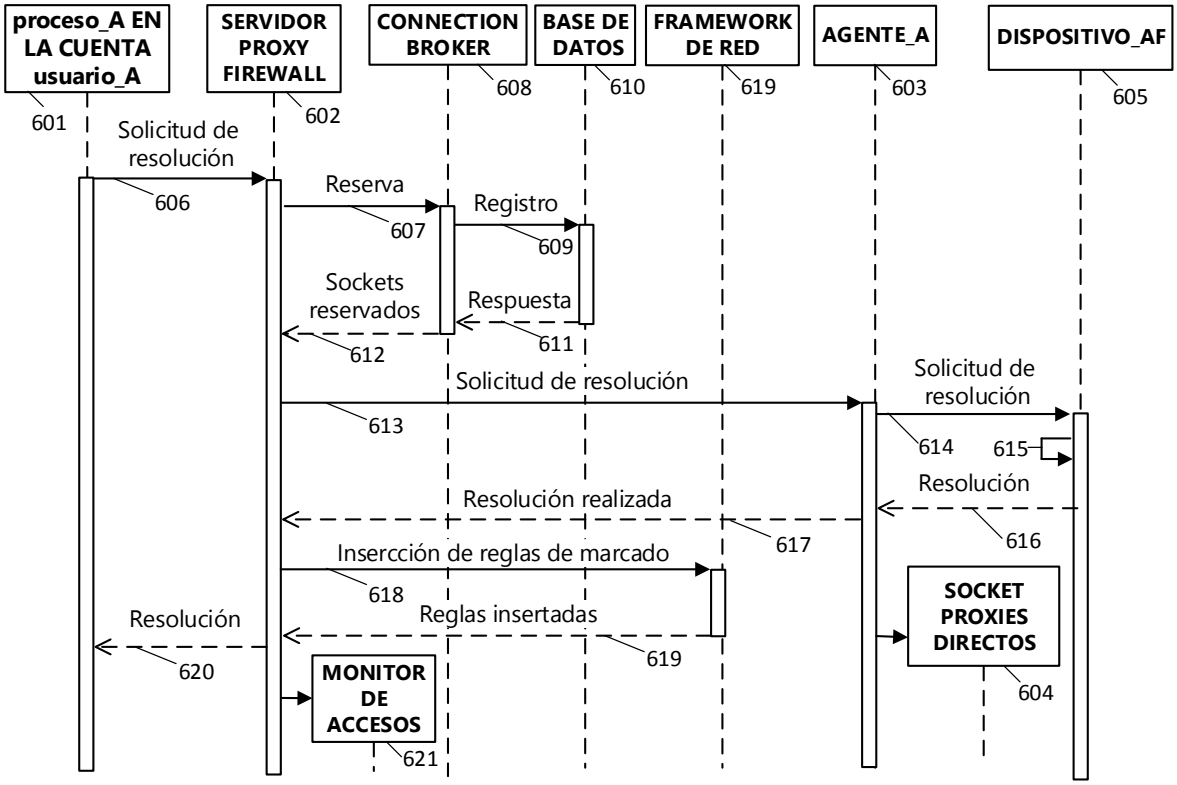


Fig. C-22: Diagrama de secuencia de IoTsafe en el que se solicita accesos a unos recursos protegidos de un dispositivo_A foráneo: Dispositivo_AF.

Aunque los *socket-proxy* directos ya se encuentran establecidos, el acceso a los mismos no es posible al ser *socket* protegidos. Para poder conceder acceso al usuario_A del proceso_A es necesario aplicar los permisos_A_U respectivos de la base de datos.

Estos permisos, son traducidos por el servidor *proxy-firewall* en reglas de marcado que se insertan (618) a través del *framework* de red. Una vez se confirma (619) este ingreso, el servidor *proxy-firewall* responde (620) la petición del proceso_A con los *sockets-proxy* directos resueltos para que pueda conectarse a ellos. Paralelamente, el servidor *proxy-firewall* instancia un monitor de accesos (621) que mantiene activos los accesos mientras esté funcionando el proceso_A.

Ejemplo 7: Ejemplo de comunicación en el que un dispositivo_AF consigue acceso a unos sockets protegidos de un dispositivo_A por proxificación directa

Este ejemplo está vinculado con el anterior: en el ejemplo 6, un dispositivo_A se conectaba con un dispositivo_AF para requerir proxificaciones directas y este ejemplo ilustra lo que ocurre en ese dispositivo_AF. Sin embargo, para mantener la coherencia de*script*iva, en el presente ejemplo ese dispositivo_AF será referido como dispositivo_A y el dispositivo que se conecta al dispositivo_A será referido como dispositivo_AF. Una vez aclarado este detalle, se procede a exponer cómo se concede en este ejemplo la proxificación directa de *sockets* protegidos a un dispositivo_AF.

La primera parte del proceso es similar a la descrita en el ejemplo 1, con la salvedad de que el dispositivo que inicia la sesión esta vez es un dispositivo_AF (en vez de ser un dispositivo_B) y que la solicitud consiste únicamente en proxificaciones directas. Para ello, El dispositivo_AF dispone de un agente_AF y de un perfil de conexión para conectarse al dispositivo_A, que le permite establecer una sesión en la que se solicita la resolución de los *socket* protegidos requeridos por el dispositivo_AF. El dispositivo_AF utiliza las credenciales del perfil_UA_{AF} para conectarse a la cuenta del usuario_A_{AF} en el sistema operativo del dispositivo_A. Esta cuenta debe tener permiso de acceso a los *socket* protegidos que se quieren proxificar. Una vez autenticadas las credenciales por el servidor *proxy-firewall* del dispositivo_A, este *software* se instancia con un *Fork* de sí mismo bajo la cuenta de usuario_A_{AF}.

Una vez llegado a este punto, el *Fork* del servidor *proxy-firewall* debe evaluar si el usuario_A_{AF} dispone de los permisos necesarios para acceder a los *socket* protegidos requeridos por el agente_AF. El procedimiento de comprobación, marcado y asignación de registro_acceso es igual al descrito en el ejemplo 2, que comprende el diagrama de secuencia de la Fig. C-10 desde (307) hasta (319). Una vez confirmados los permisos y en su caso, concedidos los accesos a los *socket-proxies* requeridos, y definidas e insertadas las reglas de marcado pertinentes, se devuelven los *socket* resueltos al agente_AF para que pueda proceder con el inicio de la proxificación directa. Para garantizar que la cuenta de usuario_A_{AF} disponga del acceso constantemente mientras dure la sesión, se instancia un monitor de accesos que periódicamente actualice todos los accesos a los *sockets-proxy* aun si no ha habido tráfico. Esto lo realizará mientras dure la sesión entre el dispositivo_AF y el dispositivo_A.

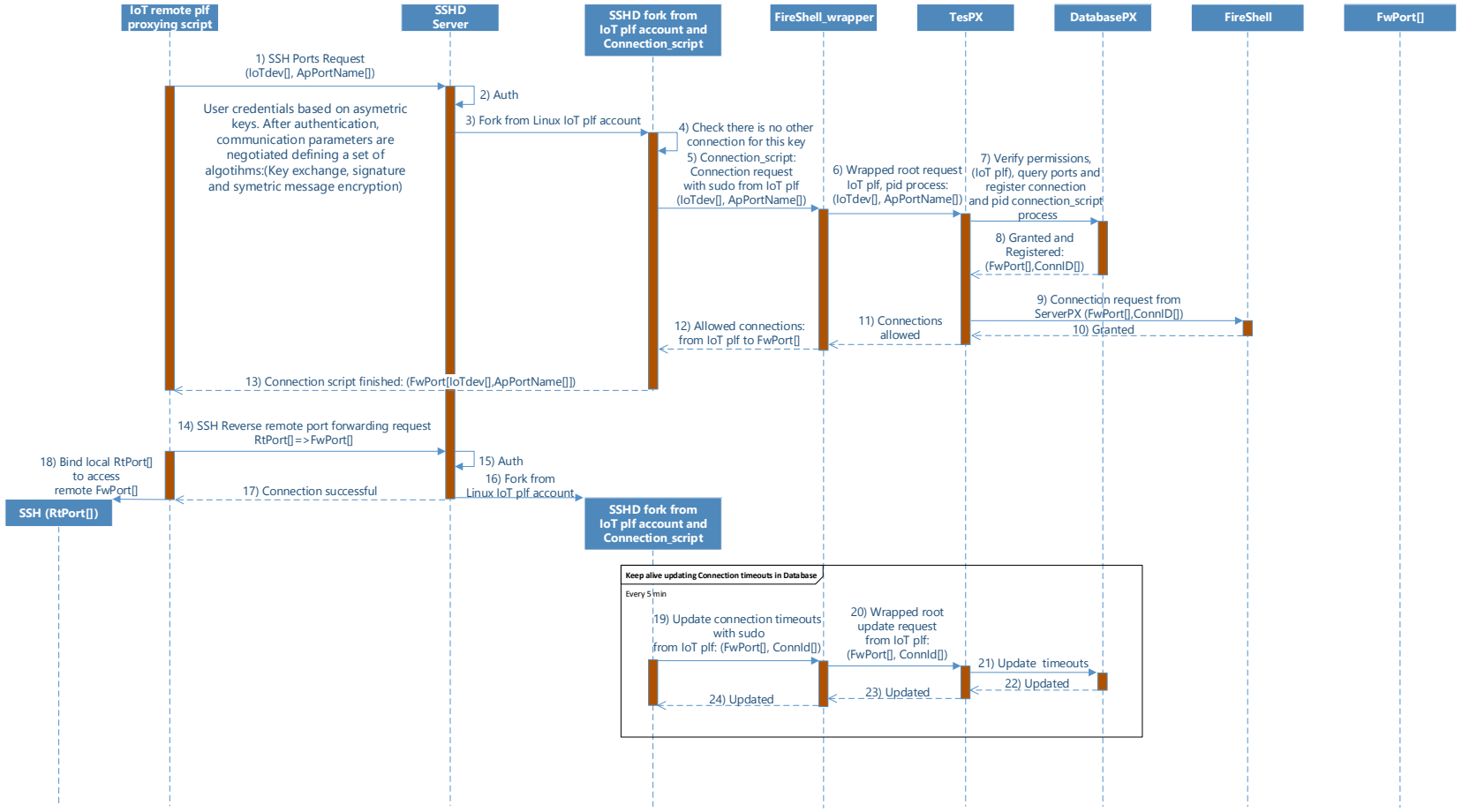


Fig. C-23: Proxificación directa de sockets protegidos de otro dispositivo_A foráneo usando la implementación preferida de IoTsafe.

Ejemplo 8: Proxificación de socket protegidos desde un dispositivo_A hacia otro dispositivo_AF de forma reversa a instancias del proceso_A del dispositivo_A

Existe la posibilidad de que una plataforma de un dispositivo_AF requiera el acceso a recursos proxificados de dispositivos_B con los que no mantiene sesión, o que esos dispositivos_B requieran de recursos de un dispositivo_A foráneo con el que tampoco mantienen sesión. En este caso, el dispositivo_A que mantiene sesión con los dispositivos_B, toma la iniciativa e inicia un proceso de proxificación reversa de puertos protegidos locales hacia el dispositivo_AF, bien por instancias de algún proceso o por algún tipo de configuración de red establecido que así lo indique. Con la proxificación reversa, el dispositivo_A pone a disposición del dispositivo_AF los *socket* protegidos que proxifique. Para ello, el agente_A establece una sesión de comunicación con el dispositivo_AF utilizando un perfil_UA_{AF} suministrado por el servidor *proxy-firewall* a instancias de un proceso_A.

En este ejemplo, el proceso_A solicita la proxificación reversa al servidor *proxy-firewall* del dispositivo_A. Al igual que en el ejemplo 2, es imprescindible que el servidor *proxy-firewall* del dispositivo_A se asegure de que el usuario_A que ejecuta el proceso_A cuente con los permisos necesarios para poder acceder a los *socket* protegidos que solicita proxificar y que, cuando corresponda, conceda y registre los accesos pertinentes y elabore e inserte las reglas de marcado oportunas. El procedimiento es idéntico al descrito en el ejemplo 2, en el diagrama de secuencias que comprende desde el (307) hasta el (319). También se requiere que el servidor *proxy-firewall* instancie un monitor de accesos que verifique que los accesos permanezcan abiertos mientras se esté ejecutando el proceso_A, actualizando periódicamente el estatus de cada registro_acceso para evitar que se cierren automáticamente por inactividad. Tras esto, el servidor *proxy-firewall* comunica al proceso_A los *socket* protegidos resueltos, de modo que cualquier proceso del usuario_A_{AF} del dispositivo_AF pueda acceder a los *sockets* protegidos del dispositivo_A a través de los *socket-proxy* reversos establecidos. Para ello, el servidor *proxy-firewall* se ha instanciado desde la cuenta de usuario_A, y por tanto puede lanzar el agente_A utilizando el perfil_UA_{AF} que inicie el procedimiento de proxificación reversa de los *socket* protegidos ya accesibles.

Ejemplo 9: Proxificación de sockets protegidos desde un dispositivo_AF hacia un dispositivo_A de forma reversa a instancias del agente_AF del dispositivo_AF

Este ejemplo 9 está muy vinculado con el ejemplo anterior. En dicho ejemplo, un dispositivo_A a instancias de un proceso_A inicia el establecimiento de proxificaciones reversas hacia un dispositivo_AF. En el presente ejemplo, se muestra lo que ocurre en ese dispositivo_AF, en el que, de nuevo, y en aras de conservar la coherencia de*script*iva, ese dispositivo_AF será referido como dispositivo_A y viceversa.

Por consiguiente, el servidor *proxy-firewall* del dispositivo_A recibe una solicitud de establecimiento de sesión de un dispositivo_AF que utiliza un perfil_UA_{AF} por la que se solicita la realización de una proxificación reversa de determinados *socket* protegidos del dispositivo_AF hacia el dispositivo_A. Esta solicitud fuerza al servidor *proxy-firewall* a instanciarse como *Fork*_A_{AF} en la cuenta de usuario_A_{AF} del dispositivo_A cuyas credenciales utilizó el agente_AF del dispositivo_AF para conectarse en el dispositivo_A. El *Fork*_A_{AF} requiere solicitar al *connection-broker* la reserva de *socket* protegidos locales en el dispositivo_A para poder realizar la proxificación reversa.

Este procedimiento es similar al seguido en el ejemplo 1 por el dispositivo_A, en el que el dispositivo_A debe reservar unos *socket* locales para las proxificaciones reversas que solicita realizar el dispositivo_B. La principal diferencia es que estos *socket* locales protegidos, resultado de la proxificación, son nuevos y no existen permisos relativos en la base de datos que den acceso a ellos. Por ello, el *connection-broker*, además de reservar esos *socket*, debe insertar registros_*socket* y permisos_A_U por cada *socket-proxy* reverso que habiliten al usuario_A_{AF} del dispositivo_A de acceso a cada *socket* resultado de la proxificación, de forma similar a como se realiza en el ejemplo 6 (607), (609), (611) y (612). Esta reserva se realiza únicamente para *socket-proxies* reversos.

Una vez reservados los *socket* e insertados los permisos correspondientes, se deben generar las reglas de marcado pertinentes e insertarlas en el sistema, para que el usuario_A_{AF} tenga acceso a esos *sockets* protegidos. El servidor *proxy-firewall* instancia un monitor de accesos para garantizar que los registros_acceso relativos a cada proxificación mantengan una inactividad baja que impida su cierre automático mientras el proceso_A siga en ejecución. El servidor *proxy-firewall*, al estar instanciado desde la cuenta del usuario_A_{AF}, forma parte del contexto de seguridad configurado al insertar las reglas de marcado y, por tanto, esta instancia dispone de acceso a los *socket* protegidos reservados en este ejemplo para la proxificación reversa. Una vez terminadas estas acciones, el servidor *proxy-firewall* procede con la proxificación reversa solicitada por el agente_AF utilizando los *sockets* reservados para ello.

Ejemplo 10: Actualización remota de elementos software

Una de las principales ventajas de IoTsafe es el desacoplo de las funcionalidades de seguridad de las aplicaciones de los dispositivos_B y su correspondiente simplificación en su diseño y mantenimiento. En los dispositivos_A, los procesos que hacen uso del sistema propuesto por IoTsafe, no requieren tampoco de *framework* de seguridad, pero sí deben incorporar en su programación el método descrito en este documento para poder comunicarse con los *socket* protegidos. No obstante, la interfaz de comunicación se basa en llamadas al sistema y marcaje de paquetes, por lo que es independiente de los protocolos de seguridad y, por consiguiente, es factible actualizar en el dispositivo_A el sistema descrito en IoTsafe, también sin afectar a las aplicaciones del dispositivo_A.

En dispositivos_A y dispositivos_B similares a los propuestos en la realización preferente de IoTsafe, es posible actualizar el agente del dispositivo_B desde el dispositivo_A sin afectar al resto del *software*. Para ello, es necesario que el dispositivo_B cuente con una versión simplificada del *proxy-firewall* que disponga del servidor *proxy stand-alone* y de una configuración adecuada en el perfil_B_A que permita al dispositivo_A utilizar un *socket-proxy* reverso para actualizar el agente del dispositivo_B. Esta estrategia puede generalizarse para proporcionar servicios de actualización a otros elementos *software* ajenos al sistema descrito por IoTsafe. En el caso particular de la actualización del cliente *proxy stand-alone*, el proceso comprende:

- Solicitar, a través de la sesión de comunicaciones establecida entre el agente_B y el servidor *proxy-firewall,* el inicio de un procedimiento de la actualización del cliente *proxy stand-alone* del agente_B por iniciativa del servidor *proxy-firewall*.
- Recibir, por parte del agente_B, la nueva versión del cliente *proxy* y actualizar el perfil_B_A.
- Probar, por parte del agente_B, la nueva versión del cliente *proxy* con el perfil_B_A actualizado estableciendo una nueva sesión de comunicaciones.
- En caso de establecerse correctamente esta nueva sesión, eliminar la versión anterior del cliente *proxy* y reemplazarlo con la nueva versión.

BIBLIOGRAFÍA

[1] C. Visual Networking Index, The Zettabyte Era: Trends and Analysis, San Jose, California, 2014.

[2] T. Grance and J. Voas, "The Internet of Things; Epic changes to follow," en *CSA Congress*, Guangzhou, 2015.

[3] K. Zhou, T. Liu and L. Zhou, "Industry 4.0: Towards future industrial opportunities and challenges," International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), Zhangjiajie, China, 2015.

[4] Cisco, "Internet of Things at a Glance," [En línea]. Disponible en: https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45-731471.pdf [Accedido: 30-09-2019].

[5] S. Ray, A. Basak and S. Bhunia, "The Patchable Internet of Things," *IEEE Spectrum,* vol. 54, no. 11, pp. 30-35, noviembre de 2017.

[6] European Cybersecurity Organisation, "Elements from ECSO members on the EU Certification Framework," octubre de 2017. [En línea]. Disponible en: http://www.ecs-org.eu/documents/uploads/elements-from-ecso-members-on-the-eu-certification-framework.pdf [Accedido: 30-09-2019].

[7] Flashpoint-intel, "Mirai Botnet Linked to Dyn DNS DDoS Attacks," 21 de octubre de 2016. [En línea]. Disponible en: https://www.flashpoint-intel.com/mirai-botnet-linked-dyn-dns-ddos-attacks/ [Accedido: 12-07-2019].

[8] M. Feily, A. Shahrestani and S. Ra, "A Survey of Botnet and Botnet Detection," en *Third International Conference on Emerging Security Information, Systems and Technologies*, Athens, Glyfada, Greece, 2009.

[9] D. Fitzgerald, "Hackers Infect Army of Cameras, DVRs for Massive Internet Attacks," *Wall Street Journal,* 30 de septiembre de 2016.

[10] C. Stupp, "Commission plans cybersecurity rules for internet-connected machines," 4 de octubre de 2016. [En línea]. Disponible en: https://www.euractiv.com/section/innovation-industry/news/commission-plans-cybersecurity-rules-for-internet-connected-machines/ [Accedido: 16-07-2019].

[11] Gartner Inc., "Gartner Says Worldwide IoT Security Spending Will Reach $1.5 Billion in 2018," 21 de marzo de 2018. [En línea]. Disponible en: https://www.gartner.com/newsroom/id/3869181 [Accedido: 10-07-2019].

[12] Directorate-General for Communications Networks, Content and Technology, "Cybersecurity Act," 19 de septiembre de 2017. [En línea]. Disponible en: https://ec.europa.eu/info/law/better-regulation/initiatives/com-2017-477_en [Accedido: 16-07-2019].

[13] European Union Agency for Network and Information Security (ENISA), "Baseline Security Recommendations for IoT," noviembre de 2017. [En línea]. Disponible en: https://www.enisa.europa.eu/publications/baseline-security-recommendations-for-iot [Accedido: 04-11-2019].

[14] The European Parliment and the Council of the European Union, "General Data Protection Regulation (GDPR)," *Official Journal of the European Union,* vol. Regulation 2016/679.

[15] Eclipse IoT Working Group, IEEE IoT, AGILE IoT, "IoT Developer Survey," 2017. [En línea]. Disponible en: https://ianskerrett.wordpress.com/2017/04/19/iot-developer-trends-2017-edition/ [Accedido: 18-07-2019].

[16] Eclipse IoT Working Group, IEEE IoT, AGILE IoT, "IoT Developer Survey," 2016. [En línea]. Disponible en: http://iot.ieee.org/images/files/pdf/iot-developer-survey-2016-report-final.pdf [Accedido: 18-07-2019].

[17] Eclipse Foundation, "IoT Developer Survey 2019 Results," Abril de 2019. [En línea]. Disponible en: https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2019.pdf [Accedido 18-07-2019].

[18] J. Ahamed and A. V. Rajan, "Internet of Things (IoT): Application systems and security vulnerabilities," en *5th International Conference on Electronic Devices, Systems and Applications (ICEDSA)*, Ras Al Khaimah, United Arab Emirates, 2016.

[19] Y. Zou, J. Zhu , X. Wang and L. Hanzo, "A Survey on Wireless Security: Technical Challenges, Recent Advances, and Future Trends," *Proceedings of the IEEE,* vol. 104, no. 9, pp. 1727 - 1765, 2016.

[20] A. R. Chordiya, S. Majumder and A. Y. Javaid, "Man-in-the-Middle (MITM) Attack Based Hijacking of HTTP Traffic Using Open Source Tools," en *IEEE International Conference on Electro/Information Technology (EIT)*, Rochester, MI, USA, 2018.

[21] E. Rescorla, "RFC8446: The Transport Layer Security (TLS) Protocol Version 1.3," Internet Engineering Task Force (IETF), agosto de 2018. [En línea]. Disponible en: https://tools.ietf.org/html/rfc8446 [Accedido: 11-09-2019].

[22] S. Deshmukh and S. S. Sonavane, "Security protocols for Internet of Things: A survey," en *International Conference on Nextgen Electronic Technologies: Silicon to Software (ICNETS2)*, Chennai, India, 2017.

[23] E. Rescorla and N. Modadugu, "RFC6347: Datagram Transport Layer Security Version 1.2," Internet Engineering Task Force (IETF), enero de 2012. [En línea]. Disponible en: https://tools.ietf.org/html/rfc6347 [Accedido: 02-01-2020].

[24] D. Adrian, K. Bhargavan, Z. Durumer, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin and P. Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," en *ACM CCS*, Colorado, 2015.

[25] W. M. Shbair, "Service-Level Monitoring of HTTPS Traffic," Université de Lorraine, 27 de noviembre de 2017. [En línea]. Disponible en: https://tel.archives-ouvertes.fr/tel-01649735/document [Accedido: 02-01-2020].

[26] C.-Y. Li, C.-C. Huang, F. Lai, S.-L. Lee and J. Wu, "A Comprehensive Overview of Government Hacking Worldwide," *IEEE Access,* vol. 6, pp. 55053 - 55073, 2018.

[27] J. Naude and L. Drevin, "The adversarial threat posed by the NSA to the integrity of the internet," en *Information Security for South Africa (ISSA)*, Johannesburg, South Africa, 2015.

[28] A. Soltani, "NSA infiltrates links to Yahoo, Google data centers worldwide, Snowden documents say," *The Washington Post,* 30 de octubre de 2013.

[29] P. Anand, "Overview of Root Causes of Software Vulnerabilities - Technical and User-Side Perspectives," en *International Conference on Software Security and Assurance (ICSSA)*, St. Polten, Austria, 2016.

[30] L. Waked, M. Mannan and A. Youssef, "The Sorry State of TLS Security in Enterprise," 24 de septiembre de 2018. [En línea]. Disponible en: https://arxiv.org/abs/1809.08729 [Accedido: 02-01-2020].

[31] T. Bradley, "Supply Chain Attacks Increase As Cybercriminals Focus On Exploiting Weak Links," *The Washington Post,* 1 de agosto de 2018.

[32] N. Philipp Tschacher, "Typosquatting in Programming Language Package," 17 de marzo de 2016. [En línea]. Disponible en: https://incolumitas.com/data/thesis.pdf [Accedido: 28-06-2019].

[33] A. Razaghpanah, N. Arian Akhavan, N. Vallina-Rodriguez, S. Sundaresan, J. Amann and P. Gill, "Studying TLS Usage in Android Apps," en *CoNEXT 2017 - International Conference on emerging Networking EXperiments and Technologies*, Seoul/Incheon, South Korea, 2017.

[34] P. Kotzias, A. Razaghpanah, K. G. Paterson, N. Vallina-Rodriguez and J. Caballero, "Coming of Age: A Longitudinal Study of TLS Deployment," en *Proceedings of the Internet Measurement Conference*, Boston, MA, USA, 2018.

[35] Y. Liu, Y. Peng, B. Wang, S. Yao and Z. Liu, "Review on cyber-physical systems," *IEEE/CAA Journal of Automatica Sinica,* vol. 4, no. 1, pp. 27-40, 2017.

[36] E. Sisinni, S. Abusayeed, S. Han, U. Jennehag and M. Gidlund, "Industrial Internet of Things: Challenges, Opportunities, and Directions," *IEEE Transactions on Industrial Informatics,* vol. 14, no. 11, pp. 4724 - 4734, 2018.

[37] S. Dong, K. Abbas and J. Raj, "A Survey on Distributed Denial of Service (DDoS) Attacks in SDN and Cloud Computing Environments," *IEEE Access,* vol. 7, pp. 80813 - 80828, 12 de junio de 2019.

[38] L. Heon Soo, "Comparative analysis of Governmental Countermeasures to cyber attacks," en *International Carnahan Conference on Security Technology (ICCST)*, Taipei, Taiwan, 2015.

[39] R. Almeida Leite, T. Gschwandtner , S. Miksch, E. Gstrein and J. Kuntner, "Visual analytics for event detection: Focusing on fraud," *Visual Informatics,* vol. 2, no. 4, pp. 198-212, 2019.

[40] I. Ahmad, T. Kumar, M. Liyanage, J. Okwuibe, M. Ylianttila and A. Gurtov, "5G security: Analysis of threats and solutions," en *IEEE Conference on Standards for Communications and Networking (CSCN)*, Helsinki, Finland, 8-20 de septiembre de. 2017.

[41] M. Törnquist, "Analysis of software vulnerabilities through," Department of Electrical and Information Technology, Faculty of Engineering, LTH, Lund University, 2017.

[42] L. Wang, J. Sushil, S. Anoop, C. Pengsu and N. Steven, "k-Zero Day Safety: A Network Security Metric for Measuring the Risk of Unknown Vulnerabilities," *IEEE Transactions on Dependable and Secure Computing,* vol. 11, no. 1, pp. 30 - 44, 2013.

[43] A. Aagaard, M. Presser and T. Andersen, "Applying Iot as a leverage for business model innovation and digital transformation," en *Global IoT Summit (GIoTS)*, Aarhus, Denmark, Denmark, 17-21 de junio de 2019.

[44] T. Lange and H. Kettani, "On Security Threats of Botnets to Cyber Systems," en *6th International Conference on Signal Processing and Integrated Networks (SPIN)*, Noida, India, India, 2019.

[45] B. Egemen Özkan and S. Bulkan, "Hidden Risks to Cyberspace Security from Obsolete COTS Software," en *11th International Conference on Cyber Conflict (CyCon)*, Tallinn, Estonia, Estonia, 28-31 de mayo de 2019.

[46] I. S. Udoh and G. Kotonya, "Developing IoT applications: challenges and frameworks," *IET Cyber-Physical Systems: Theory & Applications,* vol. 3, no. 2, pp. 65 - 72, 2018.

[47] N. Chaabouni, M. Mosbah, A. Zemmari, C. Sauvignac and P. Faruki, "Network Intrusion Detection for IoT Security Based on Learning Techniques," *IEEE Communications Surveys & Tutorials,* vol. 21, no. 3, pp. 2671 - 2701, 2019.

[48] S. Hallé, R. Khoury and B. Vignau, "10 Years of IoT Malware: A Feature-Based Taxonomy," en *IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Sofia, Bulgaria, Bulgaria, 22-26 de julio de 2019.

[49] G. Kambourakis, C. Kolias and A. Stavrou, "The Mirai botnet and the IoT Zombie Armies," en *IEEE Military Communications Conference (MILCOM)*, Baltimore, MD, USA, 2017.

[50] A. AlHogail, "Improving IoT Technology Adoption through Improving Consumer Trust," *Technologies,* vol. 2, no. 64, pp. 1-17, 2018.

[51] T. Brewster, "Russian Hackers Now Have The Power To Kill 500,000 Routers -- But The FBI Is Fighting Back," 23 de mayo de 2018. [En línea]. Disponible en: https://www.forbes.com/sites/thomasbrewster/2018/05/23/500000-routers-hacked-in-russian-plot-against-ukraine/#266e5b17310c [Accedido: 19-11-2019].

[52] B. Schneier, "Why the FBI wants you to reboot your router — and why that won't be enough next time," 6 de junio de 2018. [En línea]. Disponible en: https://www.washingtonpost.com/news/posteverything/wp/2018/06/06/why-the-fbi-wants-you-to-reboot-your-router-and-why-that-wont-be-enough-next-time/?noredirect=on&utm_term=.519a184cc266 [Accedido: 21-08-2019].

[53] A. Ghanbari, A. Laya, J. Alonso-Zarate and J. Markendahl, "Business Development in the Internet of Things: A Matter of Vertical Cooperation," *IEEE Communications Magazine,* vol. 55, no. 2, pp. 135 - 141, 2017.

[54] B. Schneier, "Patching is failing as a security paradigm," en *Click Here to Kill Everybody: Security and Survival in a Hyper-connected World*, New York, W. W. Norton & Company, 2018.

[55] G. Gianmarco Baldini and A. L. Giannopoulos, *Analysis and recommendations for a European certification and labelling framework for cybersecurity in Europe [12183/17 ADD 9],* Luxembourg: European Commission Joint Research Centre, 2017.

[56] F. Restuccia, S. D'Oro and T. Melodia, "Securing the Internet of Things in the Age of Machine Learning and Software-Defined Networking," *IEEE Internet of Things Journal,* vol. 5, no. 6, pp. 4829 - 4842, 2018.

[57] J. Valente, M. A. Wynn and A. A. Cardenas, "Stealing, Spying, and Abusing: Consequences of Attacks on Internet of Things Devices," *IEEE Security & Privacy,* vol. 17, no. 5, pp. 10-21, 2019.

[58] M. Hampson, "IoT Security Risks," 06 de septiembre de 2019. [En línea]. Disponible en: https://spectrum.ieee.org/tech-talk/telecom/internet/iot-security-risks-drones-vibrators-iot-devices-kids-toys-vulnerable-to-hacking [Accedido: 02-01-2020].

[59] J. Voas and P. A. Laplante, "The IoT Blame Game," *Computer,* vol. 50, no. 6, pp. 69 - 73, 2017.

[60] Directorate-General for Communications Networks, Content and Technology, "Cybersecurity Act," 19 de septiembre de 2017. [En línea]. Disponible en: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=COM:2017:477:FIN [Accedido: 05-12-2019].

[61] NIST, "Recommendation for Key Management," Elaine Barker, Gaithersburg, 2016.

[62] K. A. McKay, L. Bassham, M. S. Turan and N. Mouha, "DRAFT NISTIR 8114: Report on Lightweight Cryptography," agosto de 2016.

[63] C. Duan, Q. Gao, L. Wang, Q. Ma, X. Ding, H. Wang and P. Duan, "Design of an ARM9-based embedded industrial," en *IEEE Conference on Industrial Electronics and Applications*, Hangzhou, China, 2014.

[64] A. S. Haron, M. S. A. Talip, A. S. M. Khairuddin and T. F. T. M. N. Izam, "Internet of Things Platform on ARM/FPGA Using Embedded Linux," en *International Conference on Advanced Computing and Applications (ACOMP)*, Ho Chi Minh City, Vietnam, 2017.

[65] M. Kerrisk, The Linux programming interface: a Linux and UNIX system programming handbook., San Francisco: No Starch Press, 2010.

[66] USENIX Association Berkeley, "Proceedings of the 11th USENIX Security Symposium," en *USENIX Security Symposium*, San Francisco, California, USA, 2002.

[67] W. Qiang, J. Yang, H. Jin and X. Shi, "PrivGuard: Protecting Sensitive Kernel Data From Privilege Escalation Attacks," *IEEE Access,* vol. 6, pp. 46584 - 46594.

[68] R. Chandramouli, "Security Assurance Requirements for Linux Application Container Deployments," octubre de 2017. [En línea]. Disponible en: https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8176.pdf [Accedido: 28-04-2020].

[69] R. Zhang, G. Liu, X. Yuan, S. Ji and G. Zhang, "A New Intrusion Detection Mechanism in SELinux," en *International Symposium on System and Software Reliability (ISSSR)*, Shanghai, China, 2016.

[70] C. Bormann, M. Ersue, A. Keranen and C. Gomez, "RFC 7228: Terminology for Constrained-Node Networks," setiembre de 2019. [En línea]. Disponible en: https://tools.ietf.org/html/draft-bormann-lwig-7228bis-05#page-9 [Accedido: 02-01- 2020].

[71] E. M. Msahli, E. N. Cam-Winget, E. A. Serhrouchni and E. W. Whyte, "draft-msahli-ise-ieee1609-04: TLS Authentication using ITS certificate," 9 de febrero de 2020. [En línea]. Disponible en: https://tools.ietf.org/html/draft-msahli-ise-ieee1609-04 [Accedido: 03-03-2020].

[72] E. C. Kaufman, "RFC 4306: Internet Key Exchange (IKEv2) Protocol," diciembre de 2005. [En línea]. Disponible en: https://tools.ietf.org/html/rfc4306 [Accedido: 26-02-2020].

[73] S. Frankel and S. Krishnan, "RFC 6071: IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap," Febrero de 2011. [En línea]. Disponible en: https://tools.ietf.org/html/rfc2409 [Accedido: 26-02-2020].

[74] T. Ylonen and E. C. Lonvick, "RFC4252: The Secure Shell (SSH) Authentication Protocol," Enero de 2006. [En línea]. Disponible en: https://tools.ietf.org/html/rfc4252 [Accedido: 26-02-2020].

[75] M. Conti, N. Dragoni and V. Lesyk, "Attacks, A Survey of Man In The Middle," *IEEE Communications Surveys & Tutorials,* vol. 18, no. 3, pp. 2027-2051, 2016.

[76] B. Visan, J. Lee, B. Yang, A. H. Smith and E. T. Matson, "Vulnerabilities in hub architecture IoT devices," en *IEEE Annual Consumer Communications & Networking Conference (CCNC)*, Las Vegas, NV, USA, 2017.

[77] O. Salman, S. Abdallah, I. H. Elhajj, A. Chehab and A. Kayssi, "Identity-based authentication scheme for the Internet of Things," Messina, Italy, 2016.

[78] Android Open Soure Project, "SSLSocket | Android Developers," [En línea]. Disponible en: http://developer.android.com/intl/zh-cn/reference/javax/net/ssl/SSLSocket.html [Accedido: 3-06-2019].

[79] A. Popov, "RFC 7465: Prohibiting RC4 Cipher Suites," febrero de 2015. [En línea]. Disponible en: https://tools.ietf.org/html/rfc7465 [Accedido: 10-06-2019].

[80] C. Stephen Carr, "RFC 15: Network Subsystem for Time Sharing Hosts," septiembre de 1969. [En línea]. Disponible en: https://tools.ietf.org/html/rfc15 [Accedido: 29-02-2020].

[81] J. Postel and J. Reynolds, "RFC 854: Telnet Protocol Specification," mayo de 1983. [En línea]. Disponible en: https://tools.ietf.org/html/rfc854 [Accedido: 29-02-2020].

[82] J. Horwitz, "Using the Berkeley r-commands Without a Password," en *Unix System Management: Primer Plus*, Indianapolis, SAMS Publishing, 2002, p. 339.

[83] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Transport Layer Protocol," Internet Engineering Task Force, Enero de 2006. [En línea]. Disponible en: https://tools.ietf.org/html/rfc4253 [Accedido: 19-03-2020].

[84] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Connection Protocol," 2006. [En línea]. Disponible en: https://tools.ietf.org/html/rfc4254 [Accedido: 19-06-2019]

[85] S. Lehtinen and E. C. Lonvick, "RFC 4250: The Secure Shell (SSH) Protocol Assigned Numbers (Section 4.9.1)," Internet Engineering Task Force, Enero de 2006. [En línea]. Disponible en: https://tools.ietf.org/html/rfc4250#section-4.9.1 [Accedido: 13-12-2019].

[86] K. Kafle, K. Moran , S. Manandhar, A. Nadkarni and D. Poshyvanyk , "A Study of Data Store-based Home Automation," en *CODASPY '19: Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, Richardson, Texas, USA, 2019.

[87] M. Barati, A. Abdullah, N. Udzir, M. Behzadi, R. Mahmod y N. Mustapha, "Intrusion Detection System in Secure Shell Traffic in Cloud Environment," Journal of Computer Science, nº 10, pp. 2029-2036, Octubre de 2014.

[88] L. Sánchez, J. Lanza, J. R. Santana, R. Agarwal, P. G. Raverdy, T. Elsaleh, Y. Fathy, S. Jeong, A. Dadoukis, T. Korakis, S. Keranidis, P. O'Brien, J. Horgan, A. Sacchetti, G. Mastandrea, A. Fragkiadakis, P. Charalampidis, N. Seydoux, C. Ecrepont and M. Zhao, "Federation of Internet of Things Testbeds for the Realization of a Semantically-Enabled Multi-Domain Data Marketplace," *Sensors,* vol. 18, no. 10, p. 3375, 2018.

[89] P. P. Ray, "A survey of IoT cloud platforms," *Future Computing and Informatics Journal,* vol. 1, no. 1-2, pp. 35-46, 2016.

[90] R. Minerva, A. Biru and D. Rotondi, "Towards a definition of the Internet of Things (IoT)," *IEEE Internet Initiative,* no. 1, 2015.

[91] European Union Agency For Network And Information Security, "Security and Resilience of Smart Home Environments. Good practices and recommendations," diciembre de 2015. [En línea]. Disponible en: https://www.enisa.europa.eu/publications/security-resilience-good-practices [Accedido: 28-04-2020].

[92] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal and Q. Z. Sheng, "IoT Middleware: A Survey on Issues and Enabling Technologies," *IEEE Internet of Things Journal,* vol. 4, no. 1, 2017.

[93] R. Tiago Tiburski, L. Albernaz Amaral, E. de Matos, D. F. G. de Azevedo and F. Hessel, "Evaluating the use of TLS and DTLS protocols in IoT middleware systems applied to E-health," en *IEEE Annual Consumer Communications & Networking Conference (CCNC)*, Las Vegas, NV, USA, 2017.

[94] R. Bush and D. Meyer, "RFC 3439: Some Internet Architectural Guidelines and Philosophy," diciembre de 2002. [En línea]. Disponible en: http://www.ietf.org/rfc/rfc3439.txt [Accedido: 02-05-2020].

[95] R. Tiago Tiburski, L. Albernaz Amaral, E. De Matos and F. Hessel, "The importance of a standard securit y archit ecture for SOA-based iot middleware," *IEEE Communications Magazine,* vol. 53, no. 12, pp. 20-26, 2015.

[96] M. A. Razzaque, M. Milojevic-Jevric, A. Palade and S. Clarke, "Middleware for Internet of Things: A Survey," *IEEE Internet of Things Journal,* vol. 3, no. 1, pp. 70 - 95, 2015.

[97] R. T. Tiburski, C. R. Moratelli, S. F. Johann, M. V. Neves, E. Matos, L. Albernaz Amaral and F. Hessel, "Lightweight Security Architecture Based on Embedded Virtualization and Trust Mechanisms for IoT Edge Devices," *IEEE Communications Magazine,* vol. 57, no. 2, pp. 67 - 73, 2019.

[98] M. Ammar, G. Russello and B. Crispo, "Internet of Things: A survey on the security of IoT frameworks," *Journal of Information Security and Applications,* vol. 38, pp. 8-27, febrero de 2018.

[99] RERUM consortium members, "Final System Architecture, RERUM FP7-ICT-609094," 4 de septiembre de 2015. [En línea]. Disponible en: https://cordis.europa.eu/docs/projects/cnect/4/609094/080/deliverables/001-RERUMdeliverableD25Ares20153669911.pdf [Accedido: 09-06-2019].

[100] G. Moldovan, E. Z. Tragos, A. Fragkiadakis, P. H. C. and D. Calvo, "An IoT Middleware for Enhanced Security and Privacy: The RERUM Approach," en *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, Larnaca, Cyprus, 2016.

[101] G. Arfaoui, S. Gharout and J. Traoré, "Trusted Execution Environments: A Look under the Hood," en *IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, Oxford, UK, 2014.

[102] M. Sabt, M. Achemlal and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," en *IEEE Trustcom/BigDataSE/ISPA*, Helsinki, Finland, 2015.

[103] GlobalPlatform, Inc., "GlobalPlatform Device Technology: TEE Sockets API Specification," en ero de 2017. [En línea]. Disponible en: https://globalplatform.org/specs-library/tee-sockets-api-specification-v1-0-1/ [Accedido: 16-08-2019].

[104] B. McGillion, T. Dettenborn, T. Nyman and N. Asokan, "Open-TEE -- An Open Virtual Trusted Execution Environment," en *IEEE Trustcom/BigDataSE/ISPA*, Helsinki, Finland, 2015.

[105] H. Janjua, M. Ammar, B. Crispo and D. Hughes, "Towards a standards-compliant pure-software trusted execution environment for resource-constrained embedded devices," en *SysTEX '19: Proceedings of the 4th Workshop on System Software for Trusted Execution*, Huntsville, Ontario, Canada, octubre de 2019.

[106] M. Ammar, B. Crispo, J. Bart, D. Hughes and W. Daniels, "SμμV—The Security MicroVisor: A Formally-Verified Software-Based Security Architecture for the Internet of Things," *IEEE Transactions on Dependable and Secure Computing,* vol. 16, no. 5, pp. 885 - 901, 2019.

[107] J. D. de Hoz-Diego, J. Saldana, J. Fernandez-Navajas and J. Ruiz-Mas, "IoTsafe, Decoupling Security From Applications for a Safer IoT," *IEEE Access,* vol. 7, pp. 29942 - 29962, 2019.

[108] J. de Hoz-Diego, J. Saldana, J. Fernández-Navajas, J. Ruiz-Mas, R. Guerrero Rodriguez and F. J. Mar Luna, "SSH as an Alternative to TLS in IoT Environments using HTTP," en *2018 Global Internet of Things Summit (GIoTS)*, Bilbao, 2018.

[109] J. D. de Hoz Diego, "Caracterización y planificación del tráfico en una red Digital Signage," Marzo 2013. [En línea]. Disponible en: http://zaguan.unizar.es/record/10277?ln=es. [Accedido: 02-01-2020].

[110] J. De Hoz-Diego, "Secure Communication Method And System Using Network Socket Proxying". PCT Patent WO/2019/059754, 28 03 2019. [En línea]. Disponible en: https://patentscope.wipo.int/search/es/detail.jsf?docId=WO2019059754 [Accedido: 10-05-2020].

[111] A. E. Ceniceros Corral, "Memoria de estadía: Desarrollo de un prototipo de medición de parámetros ambientales y contaminantes del aire," Universidad tecnológica de Durango, Durango, 2014.

[112] V. Casas Ibarra, "Implementación del servicio de control remoto de un web browser basado en QT5 WebKit," Universidad Tecnológica de Durango, Durango, 2014.

[113] E. Ronen, A. Shamir, A.-O. Weingarten and C. O'Flynn, "IoT Goes Nuclear: Creating a ZigBee Chain Reaction," en *IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 2017.

[114] J. Solís R., "Informe de estancia profesional: Integración de un sistema Digital Signage en vehículos móviles," Universidad Tecnológica de Durango, Durango, 2016.

[115] A. Alanis Valles, M. A. Hernández Lozoya, M. I. Galindo Martínez and A. Galindo Vázquez, "Informe de estadía: Diseño, Ensamblaje e Implementación de un Sistema de Seguridad de Entrada y Salida de Personal con RFID," Universidad Tecnológica de Durango, Durango, México, 2016.

[116] B. B. Brown, «Over-the-Air (OTA) Updates in Embedded Microcontroller Applications: Design Trade-Offs and Lessons Learned,» Noviembre de 2018. [En línea]. Disponible en: https://www.analog.com/media/en/analog-dialogue/volume-52/number-4/over-the-air-ota-updates-in-embedded-microcontroller-applications.pdf [Accedido: 14-02-2020].

[117] Unión Internacional de Telecomunicaciones, "Recomendación UIT-T X.200: Tecnología de la información – Interconexión de sistemas abiertos – Modelo de referencia básico: el modelo básico," julio de 1994. [En línea]. Disponible en: https://www.itu.int/rec/T-REC-X.200-199407-I/es. [Accedido: 14-02-2020].

[118] Unión Internacional de Telecomunicciones, "Recomendación X.800: Arquitectura de seguridad de la interconexión de sistemas abiertos para aplicaciones del CCITT," 1991. [En línea]. Disponible en: https://www.itu.int/rec/T-REC-X.800/es. [Accedido: 15-02-2020].

[119] B. Hoekstra and D. Musulin, "Comparing TCP performance of tunneled and non-tunneled traffic using OpenVPN," Master Research Project, Univesiteit Van Amsterdam, System & Network Engineering, 24 de agosto de 2011. [En línea]. Disponible en: https://ext.delaat.net/rp/2010-2011/p09/report.pdf [Accedido: 18-02-2020].

[120] Unión Internacional de Telecomunicaciones, "Recomedación UIT-T X.803: Tecnología de la Información - Interconexión de sistemas abiertos - Modelo de seguridad de capas superiores," Julio de 1994. [En línea]. Disponible en: https://www.itu.int/rec/T-REC-X.803/es [Accedido: 16-02-2020].

[121] C. Wright, C. Cowan, J. Morris, S. Smalley and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," en *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, Los Alamitos, CA, USA, 2003.

[122] European Union Agency For Network and Information Security, "IoT Security Standards Gap Analysis: Mapping of existing standards against requirements on security and privacy in the area of IoT," diciembre de 2018. [En línea]. Disponible en:

https://www.enisa.europa.eu/publications/iot-security-standards-gap-analysis [Accedido: 19-04-2020].

[123] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic and S. Mangiante, "De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives," *IEEE Communications Surveys & Tutorials ,* vol. 19, no. 1, pp. 619 - 639, 2017.

[124] M. Honda, F. Huici, C. Raiciu, J. Araujo and L. Rizzo, "Rekindling Network Protocol Innovation with User-Level Stacks," *ACM SIGCOMM Computer Communication Review,* vol. 44, no. 2, pp. 52-58, April, 2014.

[125] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang and V. Paxson, "Stream Control Transmission Protocol," octubre de 2000. [En línea]. Disponible en: https://tools.ietf.org/html/rfc2960 [Accedido: 28-02-2020].

[126] E. Kohler, M. Handley and S. Floyd, "RFC 4340: Datagram Congestion Control Protocol," marzo de 2006. [En línea]. Disponible en: https://tools.ietf.org/html/rfc4340 [Accedido: 28:02:2020].

[127] J. Rosenberg, R. Mahy, P. Matthews and D. Wing, "RFC 5389: Session Traversal Utilities for NAT," octubre de 2008. [En línea]. Disponible en: https://www.ietf.org/rfc/rfc5389.txt [Accedido: 28-02-2020]

[128] R. Mahy, P. Matthews and J. Rosenberg, "RFC 5766: Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," abril de 2010. [En línea]. Disponible en: https://tools.ietf.org/html/rfc5766 [Accedido: 28-02-2020].

[129] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," abril de 2010. [En línea]. Disponible en: https://tools.ietf.org/html/rfc5245 [Accedido: 28-02-2020].

[130] A. Keranen, C. Holmberg and J. Rosenberg, "RFC 8445: Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal," julio de 2018. [En línea]. Disponible en: https://tools.ietf.org/html/rfc8445 [Accedido: 28-02-2020].

[131] B. Ford and P. Srisuresh, "Peer-to-peer communication across network address translators," en *Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, United States, 2005.

[132] Diario Oficial de las Comunidades Europeas, "Directiva 2002/58/CE del Parlamento Europeo y del Consejo relativa al tratamiento de los datos personales y a la protección de la intimidad en el sector de las comunicaciones electrónicas (Directiva sobre la privacidad y las comunicaciones electrónicas)," 12 de julio de 2012. [En línea]. Disponible en: https://eur-lex.europa.eu/legal-content/ES/TXT/PDF/?uri=CELEX:32002L0058&from=EN [Accedido: 09-01-2020].

[133] European Commision, Digital Single Market, "Proposal for a Regulation on Privacy and Electronic Communications," 10 de enero de 2017. [En línea]. Disponible en: https://ec.europa.eu/digital-single-market/en/news/proposal-regulation-privacy-and-electronic-communications [Accedido: 09-01-2020].

[134] European Union Agency for Cybersecurity (ENISA), "The Network and Information System (NIS) Directive," 06 julio de 2016. [En línea]. Disponible en: https://www.enisa.europa.eu/topics/nis-directive [Accedido: 09-01-2020].

[135] California Office of Legislative Counsel, "SB-327 Information privacy: connected devices," 28 de septiembre de 2018. [En línea]. Disponible en:

https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180SB327 [Accedido: 09-01-2020].

[136] J. D. de Hoz Diego, J. Saldana, J. Fernández-Navajas, J. Ruiz-Más, R. Guerrero Rodriguez, F. d. J. Mar Luna and R. I. Guerrera González, "Leveraging on Digital Signage Networks to Bring Connectivity to IoT Devices," en *TELCON UNI*, Lima, 2015.

[137] J. D. de Hoz Diego, J. Saldana, J. Fernández-Navajas and J. Ruiz-Mas, "Decoupling Security From Applications in CoAP-Based IoT Devices," *IEEE Internet of Things Journal,* vol. 7, no. 1, pp. 467 - 476, 2020.

[138] J. Esquerra-Soto, J. Pérez-Díaz and I. Amezcua, "Performance Analysis of 3G+ Cellular Technologies with Mobile Clients," *Instituto Tecnológico de Monterrey,* 2012.

[139] Graf, Thomas, et al, "Simple, classless Queueing Disciplines," [En línea]. Disponible en: http://lartc.org/howto/lartc.qdisc.classless.html [Accedido: 11-06-2020].

[140] A. Aijaz and A. Hamid Aghvami, F, "Cognitive Machine-to-Machine Communications," *IEEE Internet of Things Journal,* vol. 2, no. 2, abril de 2015.

[141] H. Van Styn, "Advanced Firewall Configurations with ipset," Linux Journal, 19 de marzo de 2012. [En línea]. Disponible en: https://www.linuxjournal.com/content/advanced-firewall-configurations-ipset [Accedido: 06-05-2020].

[142] S. Miano, M. Bertrone, F. Risso, M. Vásquez Bernal, Y. Lu and J. Pi, "Securing Linux with a faster and scalable iptables," *ACM SIGCOMM Computer Communication Review,* vol. 49, no. 3, pp. 1-17, 2019.

[143] M. Tumolo, "Master Thesis: Towards a faster Iptables in eBPF," Politecnico di Torino, 2018. [En línea]. Disponible en: https://webthesis.biblio.polito.it/8475/1/tesi.pdf. [Accedido: 06-05-2020].

[144] T. N. Khasawneh , M. H. AL-Sahlee and A. A. Safia, "SQL, NewSQL, and NOSQL Databases: A Comparative Survey," en *11th International Conference on Information and Communication Systems (ICICS)*, Irbid, Jordan, Jordan, 2020.

[145] I. J. Davis, M. Wexler, C. Zhang, R. C. Holt and T. Weber, "Bash2py: A bash to Python translator," en *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, QC, Canada, 2015.

[146] J. Juárez Vega, "Informe de estadía: Estudio comparativo de tecnologías y protocolos de comunicación para dispositivos del Internet de las Cosas," Universidad Politécnica de Durango, Durango, Mexico, 2016.

[147] NSA: Information Assurance Directorate, "Directorate of Capabilities Mobile Access Capability Package Version 2.0," agosto de 2017. [En línea]. Disponible en: https://www.nsa.gov/Portals/70/documents/resources/everyone/csfc/capability-packages/MA_CP_v2.0.pdf [Accedido: 18-08-2019].

[148] M. Belshe and R. Peon, "Hypertext Transfer Protocol Version 2 (HTTP/2)," mayo de 2015. [En línea]. Disponible en: https://tools.ietf.org/html/rfc7540 [Accedido: 30-06-2019].

[149] R. Seggelmann, M. Tuexen and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension," Internet Engineering Task Force, febrero de 2012. [En línea]. Disponible en: https://tools.ietf.org/html/rfc6520 [Accedido: 30-06-2019].

[150] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3: Resumption and Pre-Shared Key (PSK)," agosto de 2018. [En línea]. Disponible en: https://tools.ietf.org/html/rfc8446#page-15 [Accedido: 20-04-2020].

[151] E. Sy, C. Burkert, H. Federrath and M. Fischer, "Tracking Users across the Web via TLS Session Resumption," en *ACSAC '18: Annual Computer Security Applications*, San Juan, Puerto Rico, USA, 2018.

[152] S. Kanti Datta and C. Bonnet, "Describing Things in the Internet of Things. From CoRE Link Format to Semantic Based Descriptions," en *International Conference on Consumer Electronics*, Taiwan, 2016.

[153] F. Kaup, P. Gottschling and D. Hausheer, "PowerPi: Measuring and Modeling the Power," en *IEEE 39th Conference on Local Computer Networks (LCN)*, Edmonton, 2014.

[154] Z. Shelby, K. Hartke and C. Bormann, "RFC 7252: The Constrained Application Protocol (CoAP)," 2014. [En línea]. Disponible en: https://tools.ietf.org/html/rfc7252 [Accedido: 06-05-2020].

[155] NSA: Information Assurance Directorate, "Wireless Local Area Network Capability Package V.2.2," 26 de junio de 2018. [En línea]. Disponible en: https://www.nsa.gov/Portals/70/documents/resources/everyone/csfc/capability-packages/WLANCPv2.2_20180626.pdf [Accedido: 09-07-2019].

[156] D. Chanet, B. De Sutter, B. De Bus, L. Van Put and K. De Bosschere, "Automated reduction of the memory footprint of the Linux kernel," *ACM Transactions on Embedded Computing Systems (TECS) - Special Section LCTES'05,* vol. 6, no. 4, 2007.

[157] J.-H. Hauer and V. Handziski, "Experimental Study of the Impact of WLAN Interference on IEEE 802.15.4 Body Area Networks," en *EWSN '09 Proceedings of the 6th European Conference on Wireless Sensor Networks*, Cork, Ireland, 2009.

[158] Y. Dong, X. Youzhi and G. Mikael, "Wireless coexistence between IEEE 802.11- and IEEE 802.15.4-based networks: A survey," *International Journal of Distributed Sensor Networks,* pp. 1550-1329, julio de 2011.

[159] Chipcon, "CC24202.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver," 09 06 2004. [En línea]. Disponible en: https://inst.eecs.berkeley.edu/~cs150/Documents/CC2420.pdf [Accedido: 09-07-2019].

[160] A. Hithnawi, H. Shafagh and S. Duquennoy, "802.15.4, Understanding the Impact of Cross TechnologyInterference on IEEE," en *9th ACM international workshop on Wireless network testbeds, experimental evaluation and characterization* , Maui, Hawaii, USA, 2014.

[161] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN Systems,* vol. 17, no. 1, pp. 1-14, 1989.

[162] C. Suwannapong and C. Khunboa, "Congestion Control in CoAP Observe Group Communication," *Sensors,* 2019, 19, 3433.

[163] R. G. Lomax and D. L. Hahs-Vaughn, Statistical Concepts: A Second Course., Routledge, 2000.

[164] U. A. Chude-Okonkwo, R. Ngah and T. Abd Rahman, "Time-scale domain characterization of non-WSSUS wideband channels," *EURASIP Journal on Advances in Signal Processingvolume,* 2011.

[165] U. A. Chude Okonkwo, S. Z. Mohd Hashim, R. Ngah, N. Nanyan and T. A. Rahman, "Time-scale domain characterization of nonstationary wideband vehicle-to-vehicle propagation channel," en *IEEE Asia-Pacific Conference on Applied Electromagnetics (APACE)*, Port Dickson, Malaysia, 2010.

[166] M. Suárez-Albela, T. M. Fernández-Caramés, P. Fraga-Lamas and L. Castedo, "A Practical Evaluation of a High-Security Energy-Efficient Gateway for IoT Fog Computing Applications," *Sensors,* no. 9, p. 1978, 2017.

[167] P. M. Jacob and M. Prasanna, "A Comparative analysis on Black box testing strategies," en *International Conference on Information Science (ICIS)*, Kochi, India, 2016.

[168] GlobalPlatform Security Task Force, "Root of Trust Definitions and Requirements," marzo de 2017. [En línea]. Disponible en: https://globalplatform.org/wp-content/uploads/2018/06/GP_RoT_Definitions_and_Requirements_v1.0.1_PublicRelease_CC.pdf [Accedido: 22-04-2020].