PhD Thesis

Quarantine-mode based Live Patching for Zero Downtime Safety-critical Systems

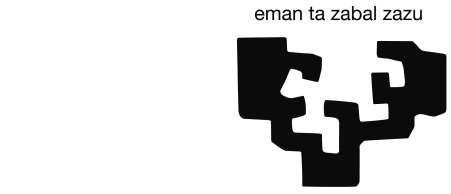
---

Imanol Mugarza Inchausti

October 2019

Supervisors:

- Eduardo Jacob Taquet
- Jorge Parra Molina



Universidad del País Vasco
Euskal Herriko Unibertsitatea

Department of Communications Engineering
Faculty of Engineering of Bilbao
University of the Basque Country UPV/EHU

PhD Thesis

Quarantine-mode based Live Patching for Zero Downtime Safety-critical Systems

---

Imanol Mugarza Inchausti

October 2019

Supervisors:

- Eduardo Jacob Taquet
- Jorge Parra Molina

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Department of Communications Engineering
Faculty of Engineering of Bilbao
University of the Basque Country UPV/EHU

Sometimes you fall before you rise
Sometimes you lose it all to find
You've gotta keep fighting
And get back up again
My champion
Oh, my champion

Alter Bridge – My Champion

ABSTRACT

Embedded systems are widely employed in consumer, industrial and military applications. Some of such systems are used in applications in which safety requirements need to be fulfilled. These systems, denoted safety-critical or safety-related systems, deal with those operational contexts whose failure or malfunction could lead to peoples death, environmental damage and/or equipment loss. To this end, functions that prevent hazardous situations or actions are carried out. Examples of safety-critical applications include transportation, medical and industrial automation applications.

Although these systems have traditionally been isolated from the open communication channels, such as the Internet, in the scope of the Industrial Internet of Things (IIoT) and/or Industry 4.0 trend, also called the fourth industrial revolution era, high inter-connectivity among sensors, machines, devices and people is foreseen. This new landscape and the increasing number of security weaknesses and flaws discovered in current industrial control systems have driven the need of implementing and adopting security measures in actual computer-based safety-critical systems. These systems need to then provide protection and resilience against cyber-security attacks and misuses, while ensuring the required safety level. The safety community has already started to address those cyber-security threats which might compromise the safe operation of the system. Furthermore, safety-related systems have often long operational cycles in which, during these periods, security requirements and objectives of the system might change.

Albeit cutting-edge security measures are adopted and incorporated during the system development, these measures may be sooner or later become out-of-date and be dodged by an attacker. Software updates are then crucial to address such security issues, which might also be desirable to upgrade other non-safety software components, such as data loggers or graphical interfaces. However, in contrast to security, stable, well-known and sound technologies are used in safety engineering. If safety risks and hazards have correctly been addressed and the operational conditions of the safety-related system are not altered, software updates are not required. Therefore, it has to be ensured that the upgrade of a non-safety related software component does not negatively impact the overall system safety and timing properties.

Commonly, a system shutdown and restart is needed to accomplish such system upgrades. Nevertheless, high-availability is frequently demanded on safety-critical systems. Software updates may not be then admissible from the business and/or service point of view when a zero downtime operation

is required. The system would be then remain vulnerable to cyber-attacks. In such case, the system would be also considered unsafe.

In this thesis, a software framework for live patching in zero downtime safety-critical systems, named *Cetratus*, is proposed, where dynamic software updates of application components are performed. The main characteristic is the quarantine-mode execution and monitoring mode, similar to the sandboxing approach, in which the new software version is executed and monitored until enough trustworthiness of the new software version is determined. This feature also provides protection against possible software and patching failures, as well as the propagation of such faults through the system. To this end, partitioning techniques are employed. Although the software upgrade is initiated by an updater, a ratification from an auditor is needed to proceed and accomplish the dynamic software updating process. These users are authenticated and logged prior proceeding with an update. The authenticity and integrity of the dynamic patch is also verified. *Cetratus* is aligned with industrial safety and security standards with respect to software updates.

Two case studies are provided. On the one hand, in the smart energy case study, a smart building electrical energy management application is examined, consisting of a Building Energy Management System (BEMS) and a Building Energy Optimization Service (BEOS) application. The BEMS monitors and controls diverse energy-related facilities on a residential building. All energy production, savings and consumption measurements are sent to the BEOS, which estimates and optimizes the overall building consumption for cost reduction and higher energy efficiency. In this case study, a live patching example is presented, where a new security layer is added to increase customer data security and privacy. More specifically, a homomorphic cryptographic algorithm is incorporated. After the update, encrypted data computed by the new software version is transmitted to the BEOS.

On the other hand, a railway case study is presented. In this case study, the Euroradio application component, the software stack of the Global System for Mobile Communications - Railway (GSM-R), is updated. The GSM-R technology is an adaptation of the Global System for Mobile (GSM) for the railway domain. The Euroradio component enables the communication among the train and track-side equipments in the European Railway Traffic Management System (ERTMS) for level two and above. In the dynamic patch, a new Message Authentication Code (MAC) scheme based on the Advanced Encryption Standard (AES) symmetry encryption algorithm is incorporated due to the security weaknesses of the former MAC scheme.

LABURPENA

Gaur egun, sistema txertatuen erabilera oso hedatua dago aplikazio industrial, komertzial eta militarretan. Hauetako batzuek segurtasun baldintza batzuk bete behar izaten dituzte; segurtasun kritikozko sistemak deritzegu, non segurtasun neurriren baten akats edo matxurak gizakumeon heriotza, ingurumenean kalteak eta/edo ekipamendu galera kausa ditzaken. Hau ekiditeko helburuz, arriskutsuak izan daitezkeen egoera eta ekintzak aurreikusten dituzten funtzioak garatzen dira. Segurtasun kritikozko sistemen erakusgarri dira transporte, medikuntza eta industria automatikarekin lotutako aplikazio ugari.

Segurtasun kritikozko sistemak, tradizionalki Internet bezalako komunikabide irekietatik isolatuak egon diren arren, geroz eta arruntagoak diren Industrial Internet of Things (IIoT) eta Industry 4.0 bezalako interkomunikazio altuko kontzeptuekin lotura handiagoa izatea aurreikusten da, makina, sentsore eta jendearen arteko komunikazio altuko ingurugiro batekin, alegia. Honekin batera, azkenaldian egungo sistema industrialetan aurkitzen ari diren segurtasun zuloek ordenagailuetan oinarritutako sistemetan segurtasun neurri berriak garatzeko beharra agerrarazi dute. Sistema hauek, beharrezko eta oinarrizko segurtasun neurriak bermatzeaz gain, ziber-erasoen eta erabilera okerreko jardueren aurkako neurriak hartzeko betebeharra dute. Segurtasun komunitatea dagoeneko hasi da sistemaren segurtasunean kalteak eragin dezaketen ziber-erasoen mehatxuaren aurkako neurrien inguruan kontzeptu eta teknologia berriak garatzen. Aipatzekoa, segurtasun kritikozko sistemek askotan luze irauten duten operazio zikloak dituztela, honek dakarren ondorioekin; ziklo hauetako bakoitzean, sistemaren segurtasun behar eta helburuak aldatu egin daitezke.

Aplikazioen garapen fasean punta-puntako segurtasun neurriak garatu eta integratu arren, momentu jakin batean iraungi egin daitezke, eraginkorrak izateari utziz eta erasotzaileari bidea erraztuz. Une horretan, segurtasun neurriak zuzendu eta eguneratu asmoz, aplikazioetan oso beharrezko bihurtzen dira software eguneratzeak. Hauetaz baliatuz, aplikazioaren beste ezaugarri batzuk aldatu daitezke, erabiltzaile interfazea, esaterako.

Normalki, sistema hauetako software eguneratze batek hau guztiz itzali eta berrabiaraztea dakar. Negozio edota zerbitzuaren ikuspuntutik, ordea, baliteke eguneratze metodo hau onartezina izatea erabilgarritasun handiko sistemetan, geldiezineko erabilgarritasun bat bermatzea kritikoa denean batez ere. Izan ere, sistema ziber erasoen aurrean zaurgarri izango litzateke, hau ez seguru bilakatuz.

Tesi honetan, geldiezinak diren segurtasun kritikozko sistemetan zuzeneko eguneraketak posible egiten dituen software framework bat proposatzen da, *Cetratus* deiturikoa, non aplikazioen osagaientzako software eguneraketa dinamikoak proposatzen diren. Ezaugarri nabarmenenak koarentena eta monitorizazio moduak dira, sandbox soluzioaren antzerakoa; bertan, software bertsio berria exekutatu eta monitorizatu egiten da gutxieneko segurtasun eta egonkortasun bat bermatu arte. Ezaugarri honek, gainera, software eta eguneraketa akatsen aurrean babes gehigarri bat ematen du, akats horien hedapena ekiditearekin batera. Hau lortze bidera, partizio teknikak erabiltzen dira. Software eguneraketa eguneratzaile batek hasten duen arren, hau bermatzen duen ikuskari baten balidazio baten beharra dago, software eguneraketa dinamikoak egiten diren momentutik. Erabiltzaile hauek aldez aurretik identifikatzen dira. Horrez gain, eguneraketa dinamikoaren autentikazio eta integritatea egiaztatzen dira. Software eguneraketei dagokienez, *Cetratus* industria segurtasun estandarrekin lerrokatuta dago.

Bi kasu azterketa ematen dira. Alde batetik, energia adimentsuaren kasuan, eraikuntza bateko energia elektrikoaren kudeaketa aplikazio bat aztertzen da, eraikinaren energia gestionatzen duen sistema (BEMS ingelesez) eta eraikinen energia optimizatzeko zerbitzua (BEOS inglesez) osagarriak dituena. BEMS-ak energiarekin lotutako instalazioak monitorizatu eta kontrolatzen ditu, bizitoki eraikuntza batean. Energia produkzio, aurrezte eta kontsumo neurketa guztiak BEOS-era bidaltzen dira, zeinek oro-harko eraikuntzaren energia kontsumoa neurtu eta optimizatzen duen, kostu murrizte eta energia eraginkortasun handiagoa lortzeko helburuz. Kasu azterketa honetan, zuzeneko eguneraketa bat aurkezten da adibide bezala, non segurtasun maila handiago bat txertatzen den erabiltzaileen segurtasun eta pribatutasun handiagoa bermatzeko. Are zehatzago, kriptografia homomorfiko bat dakarren algoritmo bat txertatzen da. Eguneraketa aplikatu ostean, bertsio eguneratuak enkriptatutako datu berriak BEOS-era transmititzen dira.

Beste aldetik, trenbide baten kasua aurkezten da. Kasu ikerketa honetan, Global System for Mobile Communications - Railway (GSM-R) aplikazio sortaren Euroradio osagaia eguneratzen da. GSM-R teknologia Global System for Mobile (GSM) trenbide domeinuaren adaptazio bat da. Euroradio osagaiak trenen eta trenekin loturiko ezaugarrien arteko komunikazioak gaitzen ditu European Railway Traffic Management System (ERTMS) sistemetan, bigarren maila eta gorakoetan. Eguneraketa dinamikoan, AES simetria enkriptatze algoritmo batean oinarritutako mezu autentikazio kode (MAC) eskema berri bat txertatzen da, aurreko eskemak erakusten dituen segurtasun akatsak direla eta.

RESUMEN

Los sistemas embebidos son ampliamente utilizados en aplicaciones de consumo, industriales o militares. Algunos de estos sistemas se emplean en aplicaciones donde se deben de cumplir requisitos de seguridad funcional. Estos sistemas, llamados sistemas-críticos, son sistemas cuyo fallo de funcionamiento puede provocar pérdida de vidas, daños graves al medio ambiente o pérdida de equipamiento. Para ello, estos sistemas realizan funciones con el objetivo de evitar situaciones y/o acciones peligrosas. Ejemplos de aplicaciones criticas incluyen las de transporte y de automatización.

Aunque estos sistemas han estado tradicionalmente aislados de los canales de comunicación habituales, como Internet, se prevé una alta conectividad entre sensores, máquinas, dispositivos y personas en el nuevo marco del Industrial Internet of Things (IIoT) y/o Industria 4.0, también llamado la cuarta revolución industrial. Este nuevo panorama y el creciente número de fallos de ciber-seguridad encontrados en los sistemas de control industriales han impulsado la necesidad de implementar y adoptar medidas de ciber-seguridad en los sistemas críticos basados en tecnología digital. Estos sistemas deben entonces, proporcionar protección y resistencia ante ataques de ciber-seguridad y abusos, mientras que garantizan el nivel de seguridad funcional necesaria. Los expertos en seguridad funcional han empezado a abordar esas amenazas de ciber-seguridad que puedan comprometer el buen funcionamiento de dichos sistemas. Además, estos sistemas suelen tener regularmente ciclos de funcionamiento largos, donde los requisitos y los objetivos de ciber-seguridad del sistema pueden cambiar.

No obstante, pese a que se incorporen medidas de ciber-seguridad punteras durante el desarrollo del sistema, estas medidas pueden quedar obsoletas tarde o temprano y ser evadidas por un atacante. Las actualizaciones de software son esenciales para abordar cuestiones relacionadas con la ciber-seguridad, que pueden ser utilizadas también para mejorar otras funcionalidades del sistema, como la interfaz gráfica. Sin embargo, a diferencia del mundo de la ciber-seguridad, se emplean tecnologías estables y bien conocidas para el desarrollo de sistemas con requisitos de seguridad funcional. Si los riesgos relacionados con la seguridad funcional y/o las condiciones de funcionamiento del sistema no cambian, las actualizaciones de software no son necesarias. Así pues, hay que asegurar que la actualización de un componente de software no relacionado con la seguridad funcional no impacte negativamente en las propiedades de seguridad funcional y temporales del sistema.

Por lo general, se require apagar y reiniciar el sistema para efectuar una actualización de software. Sin embargo, en los sistemas críticos se exige frecuentemente una alta disponibilidad. Las actualizaciones de software podrían no ser admisibles desde el punto de vista del negocio y/o del servicio cuando se requiera un funcionamiento sin interrupciones. El sistema seguiría entonces siendo vulnerable ante ciber-ataques.

En esta tesis se presenta una arquitectura y diseño de software, llamado *Cetratus*, que permite las actualizaciones en caliente en sistemas críticos, donde se efectúan actualizaciones dinámicas de los componentes de la aplicación. La característica principal es la ejecución y monitorización en modo cuarentena, donde la nueva versión del software es ejecutada y monitorizada hasta que se compruebe la confiabilidad de esta nueva versión. Esta característica también ofrece protección contra posibles fallos de software y actualización, así como la propagación de esos fallos a través del sistema. Para este propósito, se emplean técnicas de particionamiento. Aunque la actualización del software es iniciada por el usuario *Updater*, se necesita la ratificación del auditor para poder proceder y realizar la actualización dinámica. Estos usuarios son autenticados y registrados antes de continuar con la actualización. También se verifica la autenticidad e integridad del parche dinámico. *Cetratus* está alineado con las normativas de seguridad funcional y de ciber-seguridad industriales respecto a las actualizaciones de software.

Se proporcionan dos casos de estudio. Por una parte, en el caso de uso de energía inteligente, se analiza una aplicación de gestión de energía eléctrica, compuesta por un sistema de gestión de energía (BEMS por sus siglas en inglès) y un servicio de optimización de energía en la nube (BEOS por sus siglas en inglés). El BEMS monitoriza y controla las instalaciones de energía eléctrica en un edificio residencial. Toda la información relacionada con la generación, consumo y ahorro es enviada al BEOS, que estima y optimiza el consumo general del edificio para reducir los costes y aumentar la eficiencia energética. En este caso de estudio se incorpora una nueva capa de ciber-seguridad para aumentar la ciber-seguridad y privacidad de los datos de los clientes. Específicamente, se utiliza la criptografía homomórfica. Después de la actualización, todos los datos son enviados encriptados al BEOS.

Por otro lado, se presenta un caso de estudio ferroviario. En este ejemplo se actualiza el componente Euroradio, que es la que habilita las comunicaciones entre el tren y el equipamiento instalado en la vía en el sistema de gestión de tráfico ferroviario en Europa (ERTMS por sus siglas en ingles). En el ejemplo se actualiza el algoritmo utilizado para el código de autenticación del mensaje (MAC por sus siglas en inglés) basado en el algoritmo de encriptación AES, debido a los fallos de seguridad del algoritmo actual.

Acknowledgements

This work would not have been possible without the support, patience and guidance that I received from many people.

First of all, I would like to express my gratitude to my advisors Dr. Jorge Parra and Prof. Eduardo Jacob. I am grateful for their support, encouragement, time and knowledge that guided me towards the objectives of this thesis.

Many thanks to Ikerlan for the opportunity and funding that made this thesis possible. I would also like to thank my colleagues at Ikerlan, specially Jose Luis Flores, for the stimulating discussions and for cheer up and supporting me whenever I needed.

My sincere gratitude to Prof. Jean-Charles Fabre and Prof. Michael Lauer who provided me an extraordinary opportunity to join their team at LAAS-CNRS. It has been a great honour to visit the lab. Many thanks to all members at the Dependable Computing and Fault Tolerance (TSF) group for their hospitality. Special thanks to Matthieu Amy for sharing his expertise so willingly and for the provided companionship.

Lastly, I would like to warmly thank my family and friends for supporting me during these years, as well as for their unconditional love, help and encouragement.

Resumen Ejecutivo

Introducción

Los sistemas embebidos son ampliamente utilizados en aplicaciones de consumo, industriales o militares. Algunos de estos sistemas se emplean en aplicaciones donde se deben de cumplir requisitos de seguridad funcional. Estos sistemas, llamados sistemas-críticos, son sistemas cuyo fallo de funcionamiento puede provocar pérdida de vidas, daños graves al medio ambiente o pérdida de equipamiento. Para ello, estos sistemas realizan funciones con el objetivo de evitar situaciones y/o acciones peligrosas. Ejemplos de aplicaciones criticas incluyen las de transporte y de automatización industrial.

Aunque estos sistemas han estado tradicionalmente aislados de los canales de comunicación habituales, como Internet, se prevé una alta conectividad entre sensores, máquinas, dispositivos y personas en el nuevo marco del Industrial Internet of Things (IIoT) y/o Industria 4.0, también llamado la cuarta revolución industrial. Este nuevo panorama y el creciente número de fallos de ciber-seguridad encontrados en los sistemas de control industriales han impulsado la necesidad de implementar y adoptar medidas de ciber-seguridad en los sistemas críticos basados en tecnología digital. Estos sistemas deben entonces, proporcionar protección y resistencia ante ataques de ciber-seguridad y abusos, mientras que garantizan el nivel de seguridad funcional necesaria. Los expertos en seguridad funcional han empezado a abordar esas amenazas de ciber-seguridad que puedan comprometer el buen funcionamiento de dichos sistemas. Además, estos sistemas suelen tener regularmente ciclos de funcionamiento largos, donde los requisitos y los objetivos de ciber-seguridad del sistema pueden cambiar.

No obstante, pese a que se incorporen medidas de ciber-seguridad punteras durante el desarrollo del sistema, estas medidas pueden quedar obsoletas tarde o temprano. Las actualizaciones de software son esenciales para abordar cuestiones relacionadas con la ciber-seguridad, que pueden ser utilizadas también para mejorar otras funcionalidades del sistema, como la interfaz gráfica. Sin embargo, a diferencia del mundo de la ciber-seguridad, se emplean tecnologías estables y bien conocidas para el desarrollo de sistemas con requisitos de seguridad funcional. Si los riesgos relacionados con la seguridad funcional y/o las condiciones de funcionamiento del sistema no

cambian, las actualizaciones de software no son necesarias. Así pues, hay que asegurar que la actualización de un componente de software no relacionado con la seguridad funcional no impacte negativamente en las propiedades de seguridad funcional y temporales del sistema.

Por lo general, se requiere apagar y reiniciar el sistema para efectuar una actualización de software. Sin embargo, en los sistemas críticos se exige frecuentemente una alta disponibilidad. Las actualizaciones de software podrían no ser admisibles desde el punto de vista del negocio y/o del servicio cuando se requiera un funcionamiento sin interrupciones. El sistema seguiría entonces siendo vulnerable ante ciber-ataques. Se afrontan dos retos importantes que obstaculizan las actualización de dichos sistemas, que son:

- La pérdida de la disponibilidad del servicio y/o función debido a la actualización.
- Asegurar el nivel de seguridad funcional durante y después de la actualización.

El objetivo de esta tesis doctoral es la investigación y análisis de las técnicas de actualización dinámica de software para sistemas críticos. Primero, se realiza una revisión de la literatura y se evalúa el uso de los mecanismos existentes en el contexto de sistemas industriales y entornos críticos. Después de ello, se presenta una arquitectura y diseño de software que habilita las actualizaciones dinámicas de la aplicación. La solución propuesta está alineada con las normativas industriales de seguridad funcional y ciberseguridad. Finalmente, se presentan dos casos de estudio donde se intenta validar la aplicabilidad de la solución propuesta en aplicaciones reales.

Cetratus

En esta tesis se presenta *Cetratus*, una arquitectura y diseño de software que permite las actualizaciones en caliente en sistemas críticos, donde se efectúan las actualizaciones dinámicas de los componentes de la aplicación. La característica principal es la ejecución y monitorización en modo cuarentena, donde la nueva versión del software es ejecutada y monitorizada hasta que se compruebe la confiabilidad de esta nueva versión. Esta característica también ofrece protección contra los posibles fallos de software y actualización, así como la propagación de esos fallos a través del sistema. Para este propósito, se emplean técnicas de particionamiento.

Se definen dos usuarios principales, el *Updater* y el *Auditor*. El *Updater* es el encargado de desarrollar y mantener el sistema. En contraste, el

Auditor es el que verifica y valida la confiabilidad de la nueva versión del sistema, con el objetivo de garantizar la ausencia de cualquier fallo sistemático. Aunque la actualización del software es iniciada por el usuario *Updater*, se necesita la ratificación del *Auditor* para poder proceder y realizar la actualización dinámica. Estos usuarios son autenticados y registrados antes de proceder con la actualización. *Cetratus* está alineado con las normativas industriales IEC 61508 y IEC 62443 de seguridad funcional y ciber-seguridad respecto a las actualizaciones de software.

La arquitectura de la solución propuesta se divide en dos partes. Por una parte, los componentes de software específicos de la aplicación y por otra parte, los componentes de *Cetratus*, que son genéricos y re-utilizables para cualquier tipo de aplicación. Se configuran dos contenedores independientes por cada componente de aplicación, A y B respectivamente. De manera similar a la redundancia en hardware, estos contenedores, que son definidos durante la fase de diseño y desarrollo del sistema, son alternativamente definidos como el principal y el secundario. Cada contenedor se define como una partición. Las técnicas de particionamiento garantizan la independencia de ejecución tanto en el dominio espacial como temporal, mientras que a su vez posibilita la contención de posibles fallos sistemáticos. Los componentes de la aplicación pueden ejecutar y/o ofrecer diferentes tipos de funciones y/servicios, por ejemplo de ciber-seguridad, diagnóstico del sistema, pilas de comunicaciones o interfaces gráficas de usuario.

Se utiliza un mecanismo de comunicación basado en mensajería para la comunicación entre los componentes de la aplicación, así como para la comunicación entre los módulos de *Cetratus*. Para ello, se emplea un enrutador de mensajes. Este módulo también realiza tareas de duplicación y re-direccionamiento de mensajes en el modo cuarentena.

La actualización comienza con una petición del usuario *Updater* después de que se haya verificado la autenticidad e integridad del parche dinámico. En caso de que el *Auditor* apruebe dicha actualización, el sistema procede a inicializar la nueva versión de software. En esta etapa se efectúan las transformaciones del código y del estado del componente. Para ello, se comprueba cual de los contenedores A y B está siendo utilizado para el componente de la aplicación (contenedor primario). La nueva version del software se inicializa en el contener secundario.

Una vez que se efectúa la inicialización, la nueva versión del software se ejecuta y se monitoriza en modo cuarentena. En esta fase, las dos versiones del componente de la aplicación se ejecutan en paralelo, donde la información concerniente a las entradas del sistema es compartida con ambas versiones. No obstante, la salida calculada por el componente primario se establece como la salida del sistema en el modo cuarentena. Este intercambio de información se efectúa a través de mensajes internos entre los

componentes de la aplicación y los módulos de *Cetratus*, labor realizada por el enrutador de mensajes. Después de que el *Auditor* determina y valida la confiabilidad de la nueva versión del software, el sistema sustituye el componente anterior por la nueva versión. Para esta fase también se requiere la confirmación del *Auditor*. En caso de que la nueva versión no cumpla con los requisitos establecidos, tanto el *Updater* como el *Auditor* tienen la capacidad de detener y abortar dicha actualización.

Implementación y Validación

Se ha implementado un prototipo de *Cetratus* como entorno de ejecución de Ada compatible con POSIX. El uso de este lenguaje de programación está recomendado para el desarrollo de sistemas críticos. El prototipo implementado fue inicialmente integrado y validado con Real-Time Linux, ejecutándose en un ordenador de arquitectura x86. Para la validación, la solución propuesta es evaluada a través de un experimento, en el cual se proporciona una señal adicional de diente de sierra al sistema. En este punto, el sistema genera una señal triangular. El algoritmo de procesamiento de señal se ejecuta cíclicamente cada veinte milisegundos. Por medio de la actualización dinámica, se modifica el algoritmo de procesamiento de señal, suavizando la señal de salida. En vez de una señal triangular, el sistema produce una parábola. En el experimento se pre-establecieron el comportamiento de los usuarios *Updater* y *Auditor*.

En base a los resultados obtenidos, la modularidad del sistema no añade costes de computación adicionales significativos, aunque se aprecia un importante incremento en el momento de la inicialización del parche dinámico, cuando se realizan las transformaciones de código y del estado. Aun así, se necesita de doble de memoria y de tiempo de ejecución por cada componente de la aplicación (tal y como se describe anteriormente, ya que se crean dos contenedores para cada uno). También se requiere reservar tiempo de ejecución para el servicio de actualización dinámica (ofrecido por los módulos de *Cetratus*). El prototipo fue luego integrado con Integrity RTOS, un sistema operativo de tiempo real que ofrece particionamiento temporal y espacial, en un ordenador industrial x86 y en una plataforma embebida con un procesador PowerPC de 32 bits.

Casos de Estudio

En esta tesis, se proporcionan dos casos de estudio. Por una parte, en el caso de uso de energía inteligente, se analiza una aplicación de gestión de energía eléctrica, compuesta por un sistema de gestión de energía (BEMS

por sus siglas en inglés) y un servicio de optimización de energía en la nube (BEOS por sus siglas en inglés).

El BEMS, que implementa una arquitectura de criticidad mixta basado en *Cetratus*, monitoriza y controla las instalaciones de energía eléctrica en un edificio residencial. Toda la información relacionada con la generación, consumo y ahorro es enviada al BEOS, que estima y optimiza el consumo general del edificio para reducir los costes y aumentar la eficiencia energética. En este caso de estudio se incorpora una nueva capa de ciber-seguridad para aumentar la ciber-seguridad y privacidad de los datos de los clientes. Específicamente, se utiliza la criptografía homomórfica. Después de la actualización, todos los datos son enviados encriptados al BEOS.

Por otro lado, se presenta un caso de estudio ferroviario. Aunque este caso de estudio no representa una aplicación que necesariamente requiera una operación 24/7 (se puede efectuar una actualización de software ordinario al final de la misión del tren) existe interés teórico. En este ejemplo se actualiza el componente Euroradio, que es el que habilita las comunicaciones entre el tren y el equipamiento instalado en la vía en el sistema de gestión de tráfico ferroviario en Europa (ERTMS por sus siglas en inglés). En el ejemplo se actualiza el algoritmo utilizado para el código de autenticación del mensaje (MAC por sus siglas en inglés) basado en el algoritmo de encriptación AES, debido a los fallos de seguridad del algoritmo actual.

Trabajo Futuro

Aunque en esta tesis se ha propuesto un mecanismo para la actualizaciones dinámicas de software para sistemas críticos de alta disponibilidad, hace falta investigar más para poder realizar actualizaciones calientes en dichos sistemas. En primer lugar, se requieren métodos y procedimientos para verificar y validar la nueva versión. La validación depende enormemente de la aplicación desarrollada y es una tarea compleja. Hay que asegurarse de que el nuevo componente cumple con los requisitos y la funcionalidad exigida. Para este fin, se podría hacer uso de las técnicas de pruebas de regresión. Con esto en mente, el *Auditor* debería de recolectar los datos de la monitorización de la actualización de software y evaluar, por ejemplo por medio de métodos estadísticos, la nueva versión de software.

En segundo lugar, actualmente los costes de re-certificación dependen principalmente de la magnitud y complejidad del sistema a actualizar. Por consiguiente, aunque se lleven a cabo cambios mínimos en el sistema, dichos costes de re-certificación podrían aproximarse, o incluso superar, a los costes iniciales de evaluación. Es por ello necesaria la investigación en procesos de certificación modular e incremental, con el objetivo de reutilizar evidencias previamente generadas y minimizar así, los costes de la nueva certificación.

Contents

List of Figures

List of Tables

Glossary

black channel Communication channel without available evidence of design or validation according to functional safety requirements. The safety-related data is exchanged via the standard network technology..

cyber-physical system Physical mechanisms and processes monitored and controlled by software algorithms commonly executed on an embedded system.

deadlock State in which each member of a group is waiting for some other to release a given resource.

dynamic patch Differences between two program binary and/or source versions, where apart from the new executable code, additional information about the state transformation process and meta-data may be included.

embedded system A computer system specifically designed to carry out dedicated functions, usually with real-time requirements..

fieldbus An industrial computer network system for real-time distributed control.

Industry 4.0 Current trend of automation and data exchange in manufacturing technologies, based on cyber-physical system, cloud computing and Internet of Things (IoT).

MAC A security code appended to the transmitted data used to authenticate the message.

partition A strictly independent execution environment that is protected from other partitions, for which independence of execution both in the temporal and spatial domains is usually ensured.

safety Prevention of accidents and incidents which could impact on health or environmental damage.

security Protection of assets against malicious threats, misuses and abuses.

static patch Differences between two program source versions.

white channel Communication channel in which all implicated software and hardware elements are designed, implemented and validated according to the functional safety requirements.

Acronyms

ABI Application Binary Interface.

ABS Automatic Braking System.

ACL Access Control List.

AES Advanced Encryption Standard.

API Application Programming Interface.

AS Activeness Safety.

ATP Automatic Train Protection.

BEMS Building Energy Management System.

BEOS Building Energy Optimization Service.

CBC Cipher Block Chaining.

CFS Con-Freeness Safety.

CIL Common Intermediate Language.

CMAC Cipher-based Message Authentication Code.

COTS Commercial-Off-the-shelf.

DAC Discretionary Access Control.

DBT Dynamic Binary Translator.

DEMS District Energy Management System.

DES Data Encryption Standard.

DHS Department of Homeland Security.

DSU Dynamic software update.

EAL Evaluation Assurance Level.

ECB Electronic Codebook.

ECU Electronic Control Unit.

ENISA European Union Agency for Network and Information Security.

ERTMS European Railway Traffic Management System.

ESCO Energy Service Company.

ETCS European Train Control System.

EVC European Vital Computer.

GCM Galois/Counter Mode.

GHG Green House Gas.

GOT Global Offset Table.

GPS Global Positioning System.

GSM Global System for Mobile Communications.

GSM-R Global System for Mobile Communications - Railway.

GSN Goal Structuring Notation.

HMAC Hash-based Message Authentication Code.

HMI Human Machine Interface.

IACS Industrial Automation and Control System.

IAEA International Atomic Energy Agency.

IBAC Identity Based Access Control.

ICS-CERT Industrial Control Systems Cyber Emergency Response Team.

IDS Intrusion Detection System.

IEC International Electrotechnical Commission.

IIoT Industrial Internet of Things.

IMA Integrated Modular Avionics.

IoT Internet of Things.

IPS Intrusion Prevention System.

ISA International Society of Automation.

ISMS Information Security Management System.

ISO International Organization for Standardization.

IT Information Technology.

IV Initialization Vector.

JIT Just-In-Time.

JRU Juridical Recording Unit.

MAC Message Authentication Code. See *Glossary*.

MA Movement Authority.

MAC Mandatory Access Control.

MMI Man Machine Interface.

MMU Memory Management Unit.

MPU Memory Protection Unit.

NIST National Institute of Standards and Technology.

NTC National Train Control.

OSI Open Systems Interconnection.

PFD Probability of Failure on Demand.

PFH Probability of Failure per Hour.

PID Proportional Integral Derivative.

PLC Programmable Logic Controller.

POSIX Portable Operating System Interface.

PP Protection Profiles.

PR Position Report.

RBAC Role Based Access Control.

RBC Radio Block Center.

RTOS Real-Time Operating System.

SFI Software Fault Isolation.

SIL Safety Integrity Level.

TAL Typed Assembly Language.

TDEA Triple Data Encryption Algorithm.

TLS Transport Layer Security.

TMS Traffic Management System.

TOE Target Of Evaluation.

VAR Virtual Address Space.

VPC Vendor Patch Compatibility.

WCET Worst Case Execution Time.

XML eXtensible Markup Language.

1 Introduction

The embedded systems are widely used nowadays for many kinds of applications. They are broadly found in commercial, medical, industrial and military applications. In spite to a general purpose desktop computer, these systems are designed to perform specific tasks, often considering real-time and performance constraints. The use of these devices has grown exponentially during the last decade. According to *C. Ebert* and *C. Jones* [1], the worldwide market for embedded systems was around 160 billion € in 2009, with an annual growth of 9%. Besides, more than 98% of all produced microprocessors were embedded microprocessors.

One of the roles of these embedded systems in the industrial field such as automotive, railway, energy or machinery sectors, is to replace or supplement physical control mechanisms. For example, between a dozen and nearly one hundred Electronic Control Unit (ECU)s are installed today on a typical modern vehicle [2]. Moreover, one or more safety functions are carried out on these systems. These services prevent hazardous situations or actions which could impact on the safety of people and/or environment. These systems, which are defined as safety-critical systems, deal with such scenarios that whose malfunction or failure might lead to and/or equipment loss, environmental damage and/or even peoples death.

Modern safety-critical systems are becoming more and more complex, networked and distributed. As illustrated by *Feiler et al.* [3], the size and complexity of aircraft on-board software has exponentially grown in the last decades. This trend is also envisioned in the Industry 4.0 and/or Industrial Internet of Things (IIoT), also known as the fourth industrial revolution, paradigms. Moreover, even that these safety-related systems were isolated from the open communications channels, within the scope of Industry 4.0 and IIoT, the capability of sensors, machines, devices and people to be connected and communicated each other is intended. However, due to the demanded high inter-connectivity among these industrial control systems, security concerns gain importance, specially for safety-critical systems. Because of the increasing number of cyber-attacks against these systems, the safety engineering community has started to address those cyber-security threats, which can alter the proper functioning of such systems [4, 5].

1.1 Safety Vs. Security

The first significant cyber-attack compromising both safety and security properties was the Stuxnet computer virus. It was identified in 2010 and according to *Ralph Langen* it is possible to assume that the Natanz Uranium Enrichment plant in Iran was the only goal [6]. This malicious program, which was distributed locally via USB sticks and local networks, infected any computer running a Windows operating system. Nevertheless, it was targeting industrial controllers from a given manufacturer: Siemens. In order to discover them, the virus used a complex fingerprinting process, where possible available Ethernet, Profibus and MPI (Siemens proprietary communication link) interfaces of the Windows computer were tracked.

The Stuxnet computer virus was looking for the Siemens 315 (general purpose) and 417 (high-end) Programmable Logic Controller (PLC)s, which were differently attacked. On the one hand, for the 315 controller attack, during the strike condition, the execution of legitimate code simply halted. On the other hand, the legitimate code kept running at the 417 controller, but it was separated from the I/O interfaces. In fact, the 417 attack code intercepted real I/O values, but provided fake values to the legitimate program. It behaved as a fake I/O driver.

The differentiation of the safety and security terms lead to misunderstanding situations. Furthermore, some languages, such as Spanish or Swedish, provide just a single word for both concepts, which are “seguridad” and “säkerhet” respectively. Thus, neither the linguistics aids to clarify these concepts [7]. As stated by International Atomic Energy Agency (IAEA), there is not a specific distinction between the safety and security terms [8]. As clarified in this book, security tries to reduce malicious risks, prevent misuse and attacks in order to protect assets. On the contrary, safety attempts to prevent accidents and incidents which could impact on health. Safety incidents are predictable and involuntary, while security incidents are caused by on-purpose malevolent attacks or misuses [9].

In the conceptual framework proposed by [7, 10], another distinction is taken into account: where the risk is originated and where it impacts. This framework, which is depicted in Figure 1.1, aims at helping to understand the relation between safety and security concepts. As a result, the safety and security terms are divided into three notions each, where the *Defense* and *Safeguards* concepts are taken from the military and nuclear domains, respectively. These terms can be used to clarify, define, develop and asses safety and security functionalities. Three use cases are provided with the aim of capturing the differences of safety and security terms: a power grid, a nuclear power generation and finally, telecommunication and data networks [7].

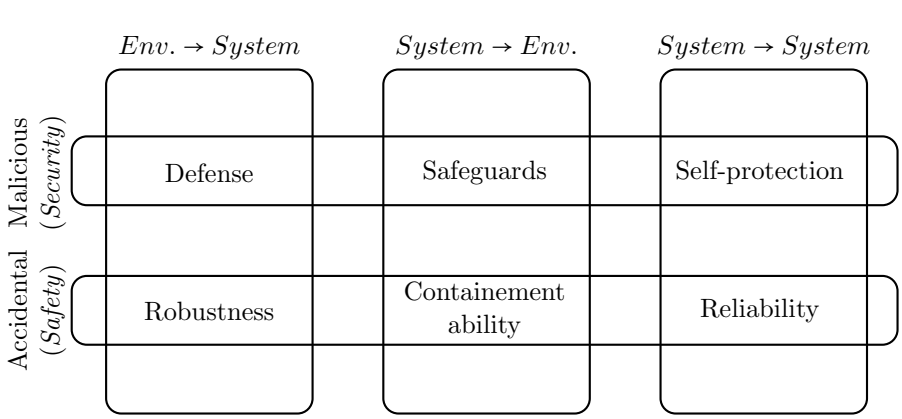


Figure 1.1: SEMA referential framework [7, 10]

However, there could be some difficulties in practice to decide when each term is applied when a system has characteristics with both safety and security connotations [11]. For example, when an Automatic Braking System (ABS) in a car is being developed and the risk of an unauthorised modification of the ROM memory contents is analysed, an unauthorised access or modification would be a security issue. Nevertheless, if this action causes fatalities, safety is also affected. In these cases, a full analysis would be required to meticulously consider the system boundaries.

1.2 Problem Statement & Main Challenges

New security flaws are discovered every day. As reported by Kaspersky lab [12], the number of vulnerabilities in Industrial Automation and Control System (IACS)s keeps growing. In 2015, 189 vulnerabilities were published, where 42% of them had medium severity and 49% were critical. Since these vulnerabilities are widely diversified among vendors and product types, it seems realistic to assume that an industrial control system will be vulnerable and could be attacked at some point while it is operating.

Due to the increasing number of security flaws and weaknesses disclosed every day, and high interconnectivity requested in current computer-based safety-critical systems, security issues come into play. Actually, security concerns have already been considered by the international safety standards, such as the International Electrotechnical Commission (IEC) 61508 [13]. As specified in the standard, "if security threats have been identified, then a vulnerability analysis should be undertaken in order to specify security requirements" (clause 7.5.2.2). Furthermore, according to the Industrial Control Systems Cyber Emergency Response Team (ICS-CERT), the num-

ber of cyber-security incidents has notably increased in the past few years. In consequence, safety-critical systems need to provide, in addition to the required safety level, resilience against cyber-security attacks and misuses.

In addition, safety-related systems have often long operational periods, up to twenty or thirty years at times, in which security requirements of the system may change. Consequently, although state-of-the-art security countermeasures are integrated while development of the system, those protection mechanisms could sooner or later become obsolete and be bypassed. Software updates are then crucial, so security issues are solved and the security trust level restored. Through this procedure, security flaws and vulnerabilities are fixed, so the possibility of a successful attack is reduced. According to IEC 62443 [14, 15], software updates are essential and shall be tested and authorised before applying them in the destination system. Opposed to security, well-known, stable and solid technologies are employed for safety. The safety trust level increases through the operational period of the system. These technologies are further tested, verified and validated through time. In case safety hazards and risks have properly been addressed or the operational conditions of the system do not change, software updates are not needed. Trust levels on safety and security software technologies through time are illustrated in Figure 1.2.

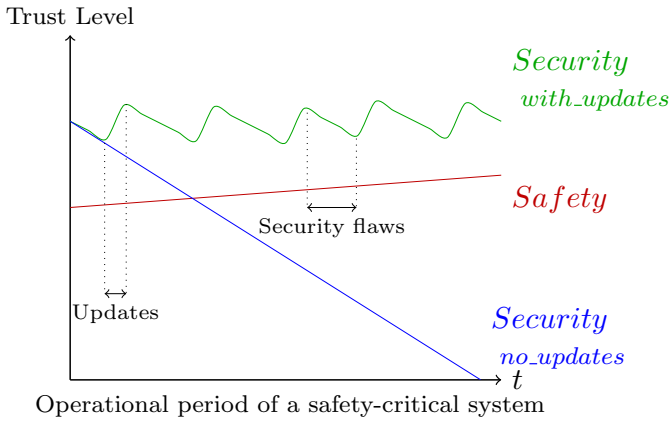


Figure 1.2: Safety & Security trust levels throughout the operational period of a safety-critical system

As it can be observed in Figure 1.2, while the safety trust level increases by itself, software updates are needed to enhance the security level when a new security bug or vulnerability is found. These upgrades may be applied regularly, as new security flaws are discovered. Nevertheless, the main challenge resides on how to perform these software updates on safety-related

systems, where it is not advised to modify the certified software once deployed, so the safety trust level is not compromised. This problem was also addressed by *Leverett et al.* [16]. Software updates might also be necessary and/or advisable for other non-safety critical software components, such as black channel communication stacks [17], Human Machine Interface (HMI)s or data loggers. Figure 1.3 depicts the group or type of upgradable software components.

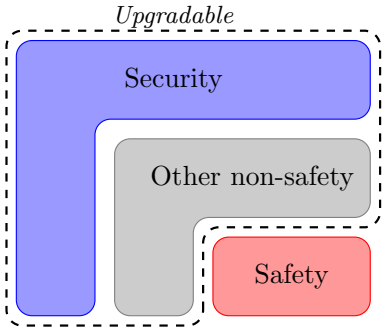


Figure 1.3: Upgradable groups/types of software components

As shown in Figure 1.4, the time period between the discovery of the vulnerability and its mitigation is defined as the window of exposure. Once the vulnerability is disclosed, mitigation strategies shall be adopted by the organizations. For this end, software patches are usually applied.

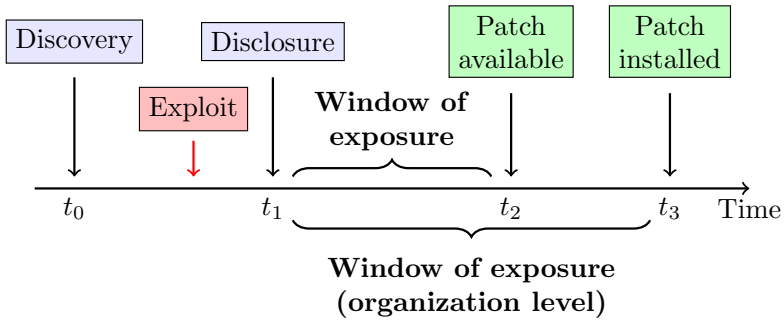


Figure 1.4: Security vulnerability life-cycle

As affirmed by *Beres and Griffin* [18], "the longer the systems stay unpatched, the bigger the risk that a vulnerability may be exploited by malicious attacks or fast spreading malware". Nonetheless, high-availability is often expected on safety-critical systems, specially in those applications where a safe state can not be reached in short time. Conventionally, a system shutdown and restart is required to perform such system upgrades.

Software updates might not be then acceptable from the service and business viewpoint when a zero downtime operation of the safety-critical system is desired. The system will still be considered vulnerable and not fulfilling anymore with the demanded security requirements. At this point, the system is considered insecure, and hence, unsafe.

In this manner, as affirmed by European Union Agency for Network and Information Security (ENISA) "the research in the area of patching and updating equipment without disruption of service and tools" is required [19, 20]. Safety and security trust levels shall be then maintained without interruption of the service. This zero downtime property makes impossible powering off the system with the aim of applying software updates. Two main challenges are then faced: the **System Shutdown** and the **Safety Assurance**.

1.2.1 System Shutdown

A system shutdown due to software updates could not be plausible when high-availability is demanded [9]. Examples of such scenarios are a nuclear reactor safety system, the Global Positioning System (GPS) or the energy supply infrastructure. One of the approaches is to test, validate and approve the new software release or patch on a replicated system within a testing benchmark, where the operational environment of the system should be closely created, as stated by the Department of Homeland Security (DHS) National Cyber security division [21]. As claimed, during these steps, it should be verified that the patched software behaves correctly. However, some environments could not be emulated on a testing benchmark due to high inter-dependencies, complexity and/or costs.

For example, if security bugs or vulnerabilities are discovered on a nuclear reactor safety system, a decision of upgrading or not the affected equipment has to be taken. On the one hand, if the protection system is decided to be updated, the whole nuclear process shall be first halted, so the software upgrade is safely applied. This might not be conceivable from the business and service point of view. In contrast, the safety integrity level would be compromised if the update is applied while operation of the reactor, because the power-off of the safety system is required. On the other hand, if the decision of not updating the system is taken, the system would remain attackable. As the system is no longer secure, it is not safe neither. Thus, the only approach which does not compromise safety is to halt the whole process, which it may not be acceptable from the service and business perspective. Note that, nuclear reactors are usually run without interruption for one or two years once the nuclear fuel is allocated.

Even though the nuclear reactor is powered off, decay heat is still pro-

duced as an effect of radiation [22]. At the time instance that the reactor is shutdown, the decay heat is approximately the 6.5 % of the previous reactor power, assuming that a steady and long power history was accomplished. Because of the decay heat, essential safety functions have to be carried out to cool down the reactor. If the decay heat is not removed, unsafe conditions of the reactor can be reached, which could lead to a nuclear disaster [22, 23]. For instance, a partial meltdown of the unit 2 on the Three-Mile Island took place after reactor shutdown because of equipment failure and operator fault. On the contrary, in Fukushima, even though the reactors were turned off after the earthquake, the tsunami incapacitated all the emergency power supply generators necessary for the reactor cooling systems. In consequence, Unit 1, 2 and 3 were meltdown [24]. Figure 1.5 shows the decay heat decrease after one year of reactor operation from the time instance in which it was taken down [25].

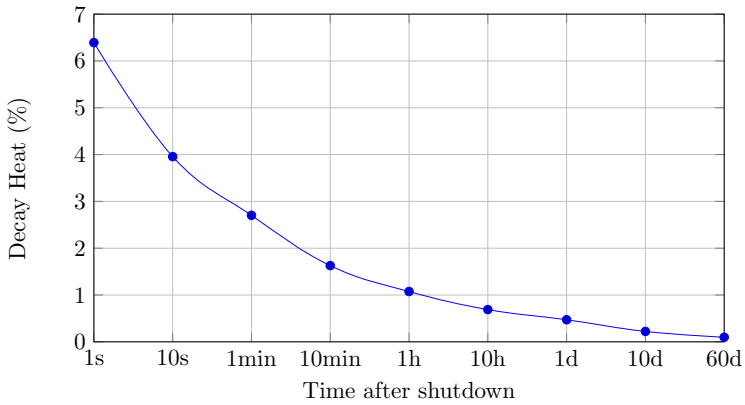


Figure 1.5: Heat decay on a nuclear reactor after shut-down

In case a cyber-attack is detected while the reactor safety systems continue being vulnerable, the scenario get worse. If the cyber-attack compromises the reactor protection systems, a nuclear accident might be unavoidable. Although the nuclear reactor is shutdown, other safety systems shall maintain essential services, such as cooling of the reactor or the backup power supply. These safety systems are then totally at the mercy of the cyber-attackers, since an appropriate operation of these systems is necessary in order to have under the reactor control. In absence of control, a nuclear disaster could be imminent. The attackers could also take down on purpose these safety-critical systems.

Likewise, ordinary software updates might not be feasible from the service's perspective when high availability is requested in energy supply infrastructure and equipment. As stated by *Khurana et al.* [26], availability is

usually a big security concern in the energy sector, since a continuous power flow is demanded and/or required. Energy control devices and systems shall then offer near 24/7 operations. High availability is also requested in the railway sector. Service disruptions, such as delays or cancelled journeys, due to software updates would lead to high expensive losses.

1.2.2 Safety Assurance

Within the implementation phases of a safety-related system, some verification, testing and validation activities are performed. The objective of these methods is to mitigate systematic failures of the system caused by errors committed in the design and coding process. Once these activities are successfully carried out and have been verified and certified by a third party, the software within the safety-related system is considered to be safe enough. Nevertheless, the safety standards do not specify how to deal with security vulnerabilities discovered once the system is operational. The modification of such software is not encouraged. Thus, a new safety validation, assessment and certification process may begin. As the security flaws which are encountered every year is increasing [12], it is possible to assume that a given safety-related system should continuously be updated to fix security issues.

Furthermore, it has to be ensured that the update of a given non-safety related software component does not impact and/or put in danger the overall system safety and/or timing properties. Other non-safety and safety-related components shall be protected both in the spatial and temporal domains against any possible malfunctions and breakdowns caused by such upgrade. In addition, according to the IEC 62443-2-3 technical report [15], IACS asset owners should "test the installation of IACS patches in a way that accurately reflects the production environment, to ensure that the reliability and operability of the IACS is not negatively affected when patches are installed on IACS in the actual production environment. Patches which have successfully passed these tests are called the authorized patches". As required by the standard [14], "a process shall exist for verifying security updates work correctly and do not introduce regressions" (Requirement SUM-1: Security update qualification).

1.3 Aim of the Thesis

The goal of the thesis is the exploration and investigation of Dynamic Software Updating (DSU) technologies for safe and secure IACSs, with the aim of formulating and suggesting a solution to the previously described problem. The designed approach shall be able to safely and securely patch the

running application software while run-time, without the need for a shut-down of the system. This mechanism would make possible the update of security-related functions while ensuring zero downtime operation. This feature would ensure the *availability*, *maintainability*, *safety* and *security* properties of the system. The dynamic software update process shall be bounded in time and in case temporal deadlines are not met, the upgrade shall be aborted. In addition, the proposed solution should try to be compliant with actual industrial safety and security standards.

1.4 Research Methodology

The research methodology defines the process to be followed in order to come upon with a solution which addresses previously described issues and achieve the aim of this research. For this purpose, the research problem is initially stated, and the challenges are then identified. After that, a state of the art on DSU techniques is elaborated, and the use of these technologies in the industrial field, especially on safety-critical systems, is evaluated. Corresponding industrial regulations are also analysed. After this, a solution to the problem is proposed based on the previously carried out research. Finally, the proposed solution is validated by means of an initial validation and two case studies: railway and smart energy.

1.5 Thesis Structure

This thesis is structured as follows:

- **Chapter 1: Introduction**: This chapter provides an introduction to the thesis, describing the problem that is being addressed and the aim of the work.

- **Chapter 2: Basic Concepts**: An overview of concepts and techniques of safety and security engineering is given in this chapter. A brief review respect to DSU concepts and techniques, as well for assurance cases, is also provided.

- **Chapter 3: State of the Art**: A literature review of safety and security standards with respect to software updates is presented. The state of the art on DSU mechanisms and systems for safe and secure IACSs is also provided. The properties of each system are also investigated.

- **Chapter 4: Cetratus**: This chapter presents and describes *Cetratus*, the proposed solution to the zero downtime safety-critical systems software patching problem.
- **Chapter 5: Validation**: This chapter provides the validation of the proposed solution. After the initial validation, two case studies are presented. On the one hand, a smart energy case study, where a smart building electrical energy management application is investigated. On the other hand, a railway case study, where the *Euroradio* component, responsible for safety-related digital communications between the train and track-side equipment, is upgraded.
- **Chapter 6: Conclusions**: Main conclusions are drawn up and possible future work is identified.

2 Basic Concepts

This chapter provides an overview of safety and security engineering techniques and technologies. Security engineering fundamentals are firstly introduced and safety engineering basics are then given. The inclusion of security tactics, techniques and approaches into the safety domain is also discussed.

2.1 Security Engineering

Security engineering deals with the design, development and manufacture of secure systems, which shall be resistant against intentional attacks and misuses [27]. Usually, a cross-disciplinary expertise is required, where mathematical knowledge, engineering skills and formal methods are implicated. Business process study, system and software engineering are also relevant. An overview of cyber-security technologies is given at the annex II of the *X.1205 - Data Networks, Open System Communications and Security* document [28]. In alignment with this document, cryptographic primitives, access control fundamentals, system integrity techniques, audit and monitoring procedures, and management concerns are analysed. Guidelines for securing IACSs are also provided by the National Institute of Standards and Technology (NIST) [29].

2.1.1 Cryptography

Cryptography is the practice and study of techniques for secure activities and operations in the presence of third parties, also defined as adversaries. In this section, basic building cryptographic primitives are described. These primitives are: symmetric stream and block ciphers, asymmetric ciphers, also called public key encryption, and one-way hash functions [27]. An overview about homomorphic cryptography is also provided. An extensive material about cryptography is given by *Alfred J. Menezes et al.* [30] and *B. Schneier* [31]. Recent advances in quantum computing have shown that certain computational problems could be solved much faster than before. Post-quantum cryptography refers to those cryptographic algorithms that are considered to be secure against an attack by a quantum computer [32]. Nonetheless, this cryptography field is not reviewed in this work.

Stream Ciphers

Stream ciphers are symmetric ciphers, where the encryption and the decryption keys are equal. In order to create the cypher-text, the plain-text is combined with a pseudo-random cipher stream. The symmetric key is used as the seed for such stream bit by bit. Figure 2.1 depicts the stream cipher-based encryption and decryption process.

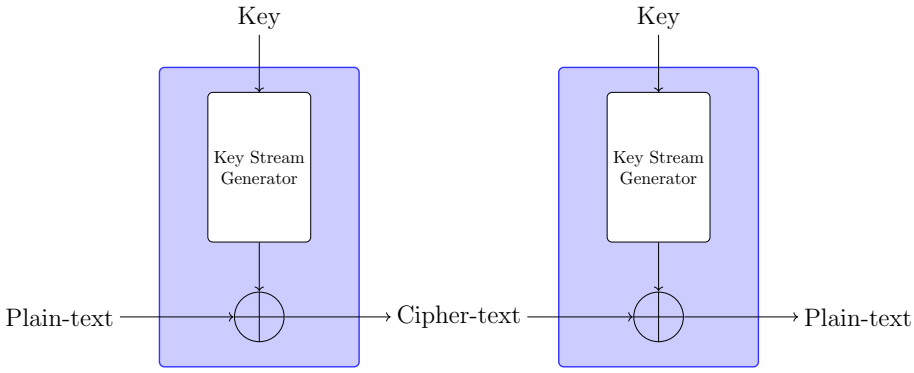


Figure 2.1: Stream cipher-based encryption and decryption

As depicted in Figure 2.1, the plain-text is combined with the key stream using the exclusive OR operation (XOR). The generated cipher-text is then decrypted through the same operation. The sender and receiver have to exactly use the same key stream to successfully perform the encryption and decryption process. If some cipher-text is lost during transmission, the synchronization of both ciphers is lost. These ciphers are usually faster than block ciphers in hardware, and have simpler circuitry.

Block Ciphers

Block ciphers are deterministic encryption/decryption algorithms which operate on fixed-length bit sizes, denoted blocks. The block size determines the length of the bit chain that will be encrypted. Both the output (cipher-text) and the input (plain-text) are the same length. Block ciphers usually support different key sizes. For example, the Advanced Encryption Standard (AES) cipher [33], which is one of the most used cryptographic algorithm, offers key sizes of 128, 192, and 256 bits. In the same way as stream ciphers, block ciphers are also symmetric algorithms.

Block ciphers can be used in different modes of operation. The block cipher mode of operation defines how a block cipher should be employed in order to provide security. A mode of operation specifies how to, repeatedly, apply the corresponding single block cipher for such plain-text data that

does not fill within a block. The simplest approach is the Electronic Codebook (ECB) mode, where the plain-text is divided into separated blocks. Padding bits are often used to fill empty slots in the last block. Each block is then independently encrypted and decrypted by the sender and the receiver, respectively. This scheme is commonly considered insecure.

Figure 2.2 illustrates the Cipher Block Chaining (CBC) mode. As depicted, in this mode, each block of plain-text is combined with the preceding cipher-text prior to its encryption. Each cipher-text block is then subjected to all previously plain-text block processed. As shown, an Initialization Vector (IV) is used at the beginning in the first block encryption process.

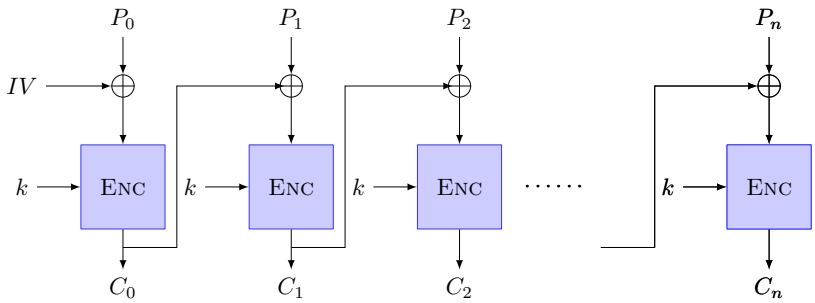


Figure 2.2: The Cipher Block Chaining (CBC) mode

The CBC mode can be applied for the computation of a Message Authentication Code (MAC), denoted Cipher-based Message Authentication Code (CMAC), which is constructed on a block cipher. In this form, the IV is set to zero. The finally computed cipher-text in the encryption procedure is used for the MAC (C_n in Figure 2.2). This last cipher-text block depends on previous blocks. This sequence ensures that any changes on the plain-text will cause the computed MAC value to change. The receiver can, consequently, verify the integrity and authenticity of the transmitted message by checking the MAC field, even though, the message is sent in plain-text. For an authenticated and encrypted communication channel, the Galois/Counter Mode (GCM) mode could be used.

Asymmetric Ciphers

Asymmetric cryptography, or public-key cryptography, is a cryptographic system that uses a pair of a private and a public keys. Through the public key encryption, a given message is encrypted with the public key. The message would be then decrypted by using the corresponding private key. The public key is usually openly distributed while the private one is securely stored by the owner. This scheme is widely employed for key exchange procedures and protocols, for example in the Transport Layer Security (TLS)

protocol. At the beginning of the communication between two parties, random numbers are exchanged by these parties by using public key cryptography. These random numbers are then used to generate session symmetric keys, as generally, asymmetric cryptography demands much higher resources than symmetric cryptography.

Public key ciphers are also used for digital signatures. In this approach, the information is signed with the private key of the sender. Afterwards, the receiver verifies such signature, and hence the authenticity of the information, by using the public key of the sender. The digital signature also ensures that the transmitted information has not been tampered or non-repudiated by any attacker.

One-way Hash functions

One-way hash functions, also known as digest functions, is a function, mathematical or otherwise, that produces fixed-length checksums (alias hash values) receiving variable-length input data. In other words, it computes a fingerprint representation of the input data. These hash functions are designed to prevent any attacker reversing the operation and recovering the original data from the hash-value.

One of the most usual uses of these functions are with digital signatures. Instead of directly signing a long message or information, such as text file, the computed hash-value is signed. In this way, computational resources are saved. Another usage, is the calculation of MACs. To this end, a hash value of the message is first calculated and then encrypted using a symmetric encryption algorithm. The length of the Hash-based Message Authentication Code (HMAC) depends on which hash function has been used for its calculation. In the same manner to a CMAC, message integrity and authenticity is provided by the HMAC. The receiver can verify the legitimacy of the received message by, once decrypting the HMAC field, checking if the provided hash value matches with the locally computed one.

Homomorphic Cryptography

Homomorphic cryptography is a cryptographic system in which computations can be performed on the cipher-text space. These operations are accomplished on encrypted data, from where the result of such calculations remains also encrypted. When decrypted, the solution matches the result of the computations as if they had been executed on the plain-text data [34]. This approach protects private information contained in the transmitted data [35]. An non-trusted party can then perform given computations on encrypted data without being aware of the included information.

2.1.2 Access control

The purpose of the access control is to regulate which principal, such as people, processes or machines, is allowed to access which resources in the system [27]. Reading, sharing and executing rules are also defined. First the principal should be authenticated, so the principal can confirm its identity claimed to the system. The system checks then the access policy, and decide if access, read or execution permissions for a given resource are granted to the authenticated principal or not (authorization).

Authentication

Authentication is the process where the identity claimed by an entity is confirmed. Different factors of authentication can be utilized for this purpose, which can be categorized into three families: something the entity *knows* (such as passwords), something the entity *has* (such as physical tokens) and something the entity *is* (such as biometrics or measurement of a human body feature). User plus password authentication is commonly adopted. Cryptography based authentication is also feasible through the use of digital certificates. According to the *X.1205 - Data Networks, Open System Communications and Security* document, authentication systems based on public cryptography are expensive. Thus, they are not widely adopted [28]. As stated by NIST, device identity and geo-localization mechanisms may also be employed to enforce security policies. Nevertheless, the authentication confidence level is not directly increased [36].

For a higher authentication confidence, two factor or multi-factor authentication schemes are usually used, which rely on a combination of several authentication factors. A successful authentication against all factors is required in order to confirm the identity of the entity. Physical tokens, for example smart cards, are typically employed for a two-factor authentication scheme, in addition to the user plus password authentication procedure. Even that the use of more authentication factors increase security, cost and complexity are also added. Consequently, an optimum trade-off has to be selected.

Authorization

The access rights to resources are specified through an access policy. The access to the resource is then granted or denied to an authenticated entity depending on these rules, which may also include read and write operations. This procedure is known as authorization. Two types of access control policies exist, which define who is the responsible of granting or denying access permissions [37]. On the one hand, access grants can be provided by

the owner of the resource. This access control type is called Discretionary Access Control (DAC). On the other hand, in Mandatory Access Control (MAC), access grants are given by the system itself, such as the operating system.

As far as the access policies are concerned, different approaches exist to define the permissions of each of the users of the system, such as Identity Based Access Control (IBAC) and Role Based Access Control (RBAC). In IBAC, permissions are granted according to the identity of the entity. An Access Control List (ACL) is usually employed, which contains a record of all identities and their access permissions. On the contrary, in RBAC, users are assigned to a role or a set of roles [38].

Firewalls

Firewalls are network security elements which monitor the incoming and out coming traffic in order to enforce a given access control policy. Three generations of firewall exist. In first generation firewalls, also known as stateless firewalls, packet filtering on the IP layer is performed. For this purpose, source and destination ports, and IP addresses are inspected. If the analysed packet does not fulfil the security rules, the packet is dropped. On the contrary, in the second generation stateful firewalls, in addition to the basic packet filtering procedures, packets are retained until enough data is available to make a judgement. For example, the stages of the TCP/IP protocol three-way handshake are observed, and packets are refused if an out of sequence of packages for the handshake is detected. In this case, transport layer protocols are examined [39].

Last generation firewalls work at the application layer, which inspect the application-related data contained within the IP packets. These are also known as layer 7 filters, referring to the last level in the Open Systems Interconnection (OSI) model. A packet classifier, such as the L7-filter[1] for the Linux kernel, could be used [40].

2.1.3 System Integrity

Malicious software (trojan horses, worms, viruses, etc) or unauthorised users might modify and/or tamper the system and data stored in it. Therefore, it shall continuously verify that the integrity of the system has not been compromised. In the Information Technology (IT) domain, antivirus software, computer programs dedicated to the detection and removal of computer viruses, and more generally, malware, is widely used. This software provides protection against cyber-threats. Signature methods can be used to

[1] `http://l7-filter.sourceforge.net/`

identify the execution of malicious code. To this end, previous knowledge concerning different malicious software is needed. As stated by the *X.1205 - Data Networks, Open System Communications and Security* technical document [28], the signature database has to be consistently up-to-date. On the contrary, behaviour methods check for illegitimate and/or suspicious behaviour of running programs.

2.1.4 Audit & Monitoring

Audit and monitoring refer to the process of checking and evaluating the overall system security. The compliance against the established security policies and criteria is regularly measured, for example access control policies, security configuration, managed security vulnerabilities or applied software updates. Systems can include servers, desktop and personal computers, network devices and IACSs. Commonly, this audit management, measurement and reporting is automatically performed. Modern operating systems already provide utilities for audit event logging.

Intrusion Detection System (IDS)s are widely employed for both network-based and host-based security incidents detection. As described by the NIST [41], these devices or software applications monitor and analyse the computer system activities and/or network traffic for signs of possible incidents. Intrusion Prevention System (IPS)s may try to, in addition to just detecting, stop recognised possible incidents. Usually, a central management server collects all audited and monitored events and reports. This information is then examined by the security administrators.

2.1.5 Management

Security management is the process of identifying the assets, analysing the security threats and implementing the corresponding security measures, procedures and policies to protect them from security-related threats and vulnerabilities. Assets may make reference to data, IACSs, people, etc. Different methods to evaluate such security risks exist. Four risk analysis methods were analysed by *Syalim et al.* [42]. The investigated techniques were: Mehari, Magerit, NIST SP800-30 and Microsoft's Security Management Guide. Depending on the severity of the identified risks, asset owners should take the appropriate countermeasures to mitigate them.

Configuration management allows verifying and modify (if needed) the security-related configuration options of the devices. Access control policies, such as firewall rules, might be specified. Access rights for entities (system users, applications, services or other devices) could be limited. Organizations should also define and establish methods and procedures for patch management, which as stated by the NIST [43], "is the process for

identifying, acquiring, installing, and verifying patches for products and systems". An effective and efficient patch management is essential to achieve and maintain a sound security through the systems and products lifecycle.

2.2 Safety Engineering

In order to design, develop and maintain safety-related systems, functional safety technologies and engineering methods are employed. A guide to functional safety is provided by *Smith* and *Simpson* [44]. Because of the wide spectrum of the safety engineering field, this section provides a brief summary of some safety concepts and technologies. The following topics are outlined: *Reliability*, *Partitioning*, *Software Techniques* and *Security for Safety*.

2.2.1 Reliability

Reliability is defined as the ability or probability of an item to provide the required function under given conditions for a given time interval [45, 46]. An item can be a single component, a group or a system composed by several components. The reliability block diagram is first designed, which reflects how each element contributes to a system failure. Parallel (redundant) and series configuration are usually employed. After that, the operating conditions and the failure rate for each element are determined. This information can be obtained in several ways, for example from historical data, government and commercial data or testing. Finally, the reliability for each element is calculated and the system reliability is computed, denoted $R_S(t)$. Common cause failures shall also be taken into account, where failure links and dependencies are analysed.

In safety engineering, the failure rate λ of elements and the Probability of Failure on Demand $(\text{PFD})_{avg}$ and Probability of Failure per Hour (PFH) values of the system are calculated [44]. The probabilities of failure and reliability are antagonisms measures. The first one measures how often the system fails, while the second one quantifies the probability of the system to provide the required function. More detailed information, data, procedures and calculations are provided by *Alessandro* [45], *Rausand et al.* [46], *Shooman* [47], and *Smith* and *Simpson* [44].

2.2.2 Partitioning

In order to assure that the specification, design, implementation, validation and certification (if needed) stages are independent among the mixed-criticality software components, partitioning is used. Partitioning prevents

applications interfering each other, except if not specifically designed for it. A partition is a strictly independent execution environment that is protected from other partitions. It ensures the separation of information of differing sensitivity or criticality levels, which can alter the correct operation or real-time performance of the system [48, 49]. These techniques are applicable for both safety and security domains. In safety, the objective is fault containment, so it does not propagate through the system. On the contrary, in security, the consequences of misuse or malicious intrusions are kept under control. The possible faults and intrusions are encapsulated or contained within a partition.

Two approaches exist for partitioning: operating system based or executive kernel based. In the first approach, the operating system distributes client processes among partitions, where the isolation is obtained by enhancing the host operating system's features so that partitioning techniques can be implemented. The *INTEGRITY* RTOS developed by *Green Hills Software*[2] is an example of such approach. In contrast, an executive kernel offers a virtualization layer, where several virtual machines with their own operating systems could be independently executed, for example in Xtratum [50, 51]. Figure 2.3 depicts an example of partitioned Integrated Modular Avionics (IMA) architecture given by *Blasch et al.* [52] based on an executive kernel:

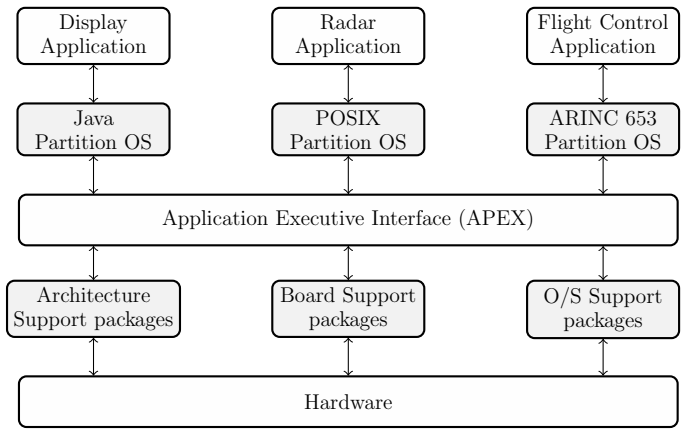


Figure 2.3: Example of an IMA architecture [52]

Partitioning is used to integrate on the same computing platform critical and non-critical applications. These systems are often called mixed-criticality systems [53]. However, as stated by the IEC 61508 standard [13], independence of execution both in the spatial and temporal domains

[2] `https://www.ghs.com/products/rtos/integrity.html`

shall be achieved and demonstrated [49]. Spatial and temporal partitioning concepts explained by *Rushby* [49] are next summarized.

Spatial partitioning

Spatial partitioning ensures that the software within a partition can not access the resources, such as code, data or private devices, of another partition. Hardware mechanisms such as Memory Management Unit (MMU) and Memory Protection Unit (MPU) are usually used. These techniques prevent to an application hosted within a partition access and write to a memory belonging to another one. The MMU or MPU tables are managed by the operating system or the executive kernel. This software layer is also protected itself, so applications running in top of it can not modify it. As an alternative, Software Fault Isolation (SFI) techniques might be used [54].

Temporal partitioning

Temporal partitioning ensures that the activities carried out within a partition do not compromise the timing properties of other partitions. The main matter is to avoid an application denying service to other partitions, such as monopolizing the CPU or, crashing or halting the system. Schedule overruns, for example when a given partition takes longer time to perform its activities, and runaway executions, such as infinite loops, have to be also avoided. Two approaches to achieve temporal partitioning exist. On the one hand, a two-level structure in which first the kernel only schedules partitions. The application running in each partition is then responsible of scheduling the desired tasks. On the other hand, in second single-level structure approach, the kernel directly schedules applications tasks. Applications are then executed only during the time slices they are assigned to.

2.2.3 Software Techniques

In order to achieve and ensure the required functional safety level, different software techniques are employed. These techniques are used for the design, implementation and validation of all safety-related software, including application software, operating systems, communication stacks, HMI software, in addition to firmware. The overall goal of these methods and/or guidelines is the avoidance and control of systematic system faults and failures.

Table 2.1 provides the recommended software techniques for software architecture and design described by the IEC 61508 [13] standard for different Safety Integrity Level (SIL)s, where R, HR and NR denote *Recommended*, *Highly Recommended* and *Not Recommended*, respectively.

ID	Technique	SIL 1	SIL 2	SIL 3	SIL 4
1	Fault detection	–	R	HR	HR
2	Error detecting codes	R	R	R	HR
3a	Failure assertion programming	R	R	R	HR
3b/c	Diverse monitor techniques (independence between the monitor and monitored function on the same computer)	–	R	R	–
3b	Diverse monitor techniques (with separation between the monitor and monitored computer)	–	R	R	HR
3d	Diverse redundancy	–	–	–	R
3e	Functionally diverse redundancy	–	–	R	HR
3f	Backward recovery	R	R	–	NR
3g	Stateless software design	–	–	R	HR
4a	Re-try fault recovery mechanism	R	R	–	–
4b	Graceful degradation	R	R	HR	HR
5	Artificial intelligence - fault correction	–	NR	NR	NR
6	Dynamic reconfiguration	–	NR	NR	NR
7	Modular approach	HR	HR	HR	HR
8	Use of trusted/verified software	R	HR	HR	HR
9	Forward traceability between requirements and architecture	R	R	HR	HR
10	Backward traceability between requirements and architecture	R	R	HR	HR
11a	Structured diagrammatic methods	HR	HR	HR	HR
11b	Semi-formal methods	R	R	HR	HR
11c	Formal design and refinement methods	–	R	R	HR
11d	Automatic software generation	R	R	R	R
12	Computer-aided specification and design tools	R	R	HR	HR
13a	Cyclic behaviour	R	HR	HR	HR
13b	Time-triggered architecture	R	HR	HR	HR
13c	Event-driven	R	HR	HR	–
14	Static resource allocation	–	R	HR	HR
15	Static synchronization of access to shared resources	–	–	R	HR

Table 2.1: Recommended software techniques by the IEC 61508 [13] for software architecture and design

As far as safety-related communications are concerned, the IEC 61508 uses the concepts of the so called black channel or white channel approaches to define the requirements of the base fieldbuses for transmission of safety data. Whether a communication channel is white or black is determined by where the safety measures are accomplished with respect to the base fieldbus and/or communication channel. The IEC 61784-3 standard [17] specifies the functional safety communication profiles that use the black channel approach. Figure 2.4 shows the black channel approach.

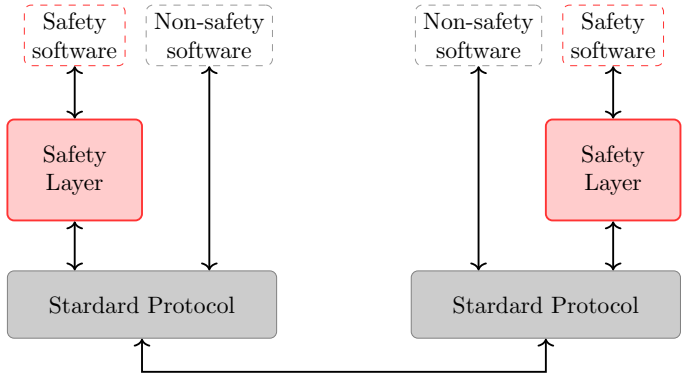


Figure 2.4: The black channel approach

As shown in Figure 2.4, the safety-related data is transmitted via standard networking elements. To this end, a safety layer implements the required measures to ensure the integrity of the data across the communication link. Countermeasures against data corruption, unintended repetition and incorrect sequence delays (among others) are taken over the standard communication protocol. In contrast, Figure 2.5 depicts the white channel approach.

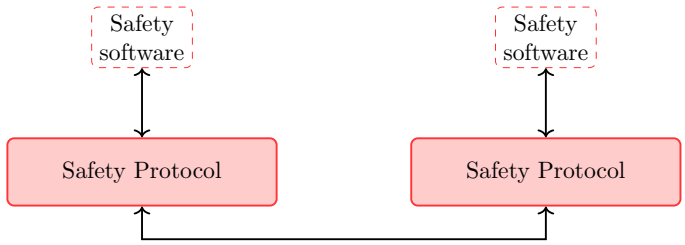


Figure 2.5: The white channel approach

On the contrary to black channel approach, in the so called white channel approach, all the hardware and software elements involved in the safety-related digital communications shall be designed, implemented and validated according to the functional safety requirements (safety protocols).

2.2.4 Security for Safety

At the hazard and risk analysis phase, hazards, hazardous events and hazardous situations are agreed so the risks associated with those events can be determined. Due to the security threats, this evaluation needs to be extended in case malevolent or unauthorised actions are identified. During this security threat analysis, deliberate misuse, vandalism and criminalism are taken into account [44]. After that, security requirements are defined for the safety-critical system.

An integrated safety and security co-engineering is fundamental for the design, development and maintenance of safe and secure mixed-criticality systems. As claimed by *Hunter* [55], "ensuring continued alignment of the dependence and compatibility of safety and security through the lifecyle by is the key to their integration". This topic was also analysed at the ITEA 3 MERgE project [4, 5, 56]. As part of it, an extensive literature review on safety and security co-engineering of software intensive critical information systems was carried out. An overview of methods, techniques and tools which have been adjusted in order to cover both safety and security aspects is also presented [56, 57]. Utilities and procedures coming from the safety engineering were revised and arranged for the security one, and vice versa. A broad state of the art on industrial safety and security standards is also provided. A survey of approaches for IACS risk assessments and design involving both safety and security risks was provided by *Kriaa et al.* [58]. As discussed, the trend is to keep safety and security activities separated from each other. As stated by *Hunter* [59], the safety and non-safety boundaries have to be established, and isolation among such functions ensured. This could be achieved through partitioning techniques.

The combination of safety and non-safety-critical applications, such as security, is defined as a mixed-criticality system, where a strong isolation among applications is crucial [53]. In safety, the objective is fault containment, where the propagation of the fault through the system is prevented. In security, the consequences of misuse or malicious intrusions are kept under control. Possible faults and intrusions are confined through partitioning. A mixed-criticality architecture for safety and security was proposed by *Bock et all.* [60]. An onion approach was considered, where a security shell is placed between the network and the safety application layer.

The onion skin approach is also applicable for safe and secure digital communications. As stated by the IEC 61784-3 technical standard [17], "when an application requires electronic security measures, the security shall be implemented within the black channel". According to *Åkerberg et al.* [61], wired fieldbus protocols have often incorporated safety measures to conform functional safety requirements, while, on the contrary, security coun-

termeasures are usually provided in wireless technologies as a result of high accessibility to the communication media.

Figure 2.6 shows the onion skin design approach for mixed-criticality safe and secure systems.

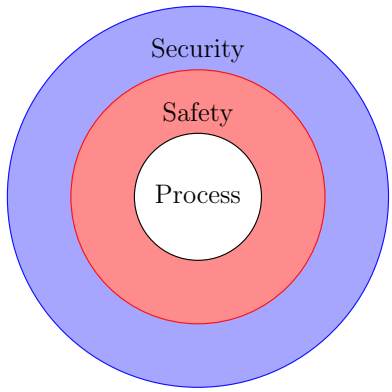


Figure 2.6: Onion skin approach for mixed-criticality safe and secure systems

An approach to secure an industrial fieldbus protocol was presented by *Wieczorek et al.* [62], where a stream cipher and a MAC algorithm were employed. Some initial runtime analyses were performed on proof-of-concept implementation for the EtherCAT fieldbus communication protocol. Furthermore, a framework for wired and wireless communications addressing both functional safety and security was proposed by *Åkerberg et al.* [61]. For this purpose, a security layer is introduced between the *Safety Layer* and the *Standard Protocol* ones shown in Figure 2.4. Following the black channel principle and the onion skin approach shown in Figure 2.6, the safety-related data is recursively encapsulated and protected. Initially, the information is protected against unintentional and/or random communication failures. To this end, safety measures are adopted in the *Safety Layer*. After that, end-to-end integrity and authentication, as similarly proposed by *Wieczorek et al.* [62], is then provided by the security layer.

2.3 Assurance Cases

Assurance cases (safety or a security cases) is defined as an evidences supported argumentation to justify that the system is safe or secure enough to operate in a given context. This information is then supplied to a certification body. The main elements of a safety or security case are requirements, evidences, arguments, and context. The safety case shall define a safety goal, which usually is the compliance of the system or product against given

safety or security standards and/or requirements. The main safety or security goal is repetitively divided into safety sub-goals. In the end, these sub-goals shall be connected to a given set of claims. These claims should be sustained and justified by clear evidences. The context of the safety or security goal shall be clearly stated, since any safe system might behave unsafely if inappropriately used [63]. Even that this argumentation approach is usually adopted for safety, the same strategy can directly be employed for security. This is the reason why this thesis refers to safety and security cases instead of safety ones exclusively. Security cases were presented and used by *Goodenough et al.* [64], *Graydon and Kelly* [65], *He and Johnson* [66] and *Preschern* [67].

Nowadays, safety and security cases are usually reported through the use of the Goal Structuring Notation (GSN). GSN, which was introduced by *Kelly* [63, 68, 69], is a graphical argumentation notation, where requirements, evidences, arguments, the context of the safety case and the links among them are graphically represented. A preliminary safety case, which is created in the early phases of the system development, of a distributed computing platform for an aero-engine control was provided by *Kelly et al.* in [69]. Specific architectural level random and systematic (both timing and functional) failures modes were taken into account. Quantitative and qualitative argumentation, based on claims and conclusively supported by evidences, is depicted. This argumentation justifies the adopted safety mechanisms so the considered failure modes of the system are kept under control, and the system behaves safely enough. Timing behaviour correctness of the system is also verified. Nevertheless, a more extensive software timing argumentation for a computer-assisted braking system was constructed by *Graydon and Bate* [70]. A top-level safety argumentation is firstly provided. After that, safety argumentations over Worst Case Execution Time (WCET) estimations and timing analysis are depicted. Moreover, a modular safety case for a generic hypervisor was presented by *Larrucea* [71]. The minimum reasonable safety evidences and argumentations for this software component are described. The modularity strategy permits the reuse of safety case elements and components.

On the contrary, a similar argumentation methodology is employed for security. A security assurance case arguing which security countermeasures are necessary in order to avoid common coding defects is presented by *Goodenough et al.* [64]. More specifically, a partial security assurance case is given, where claims and evidences to justify that the system is free of buffer overflow coding defects is illustrated. Results gathered at the developing process of the system, for example from static analyses or robustness testing, are presented as evidences. For this purpose, the GSN representation is used. This graphical notation is also used by *He and Johnson* [66]

to justify that the adopted cyber-security policies and procedures on complex healthcare organisations provide a sound protection and confidentiality level. In a similar way to safety, security cases can be used to prove the conformity of the system against security standards, as done by *Graydon and Kelly* [65] and *Preschern* [67], where the security tactic goals are linked to the Common Criteria [72] framework through the use of GSN.

As far as the development process and maintainability of a safety and security case are concerned, a progressive development approach is usually chosen to elaborate it [63]. As stated by *Goodenough et al.* [64], "developing even the preliminary outlines of an assurance case as early as possible in the software development life cycle can lead to improvement in the development process". This is due to the fact that the developers can then focus their attention on those concerns and matters to resolve. As indicated by *M. Nicholson et al.* [73], an assurance case might be incrementally maintained because of system changes. In fact, from the security perspective, some security standards introduce objectives and requirements concerning firmware or software upgrades, such as in Common Criteria [72, 74]. The maintainability of a safety case for an IMA is analysed by *M. Nicholson et al.* [73]. As stated, an incremental certification, which has been already used in large and complex platforms, enables the qualification of new applications while the existing one is maintained. A re-evaluation of the whole system would not be necessary. However, only safety considerations are contemplated. Security changes are more susceptible to happen.

2.4 Dynamic Software Updates

A Dynamic Software Update (DSU) consist on updating a computer program while it is being executed without the need of a restart. These techniques improve system uptime and availability, which is an attractive feature for mission-critical or safety-critical systems, such as the air traffic control system or the telephone switches. When a software update is desired, the new code is first loaded and the actual state transformed then into a new one, which should be understandable by the new program. At this point, the new program is ready to be run. Nevertheless, an acceptable update availability must be guaranteed during this process for those applications where time constraints are present.

The execution of a computer program can be considered as a tuple (P, δ), where P is the program code and δ is the current program state. The current program state δ can include the state maintained by the operating system for the program P (such as file descriptors or open network connections), the heap (where global variables are stored), stack frames and program counters. In contrast, the program code P includes a set instructions executable by the

system. The DSU mechanism transforms the actual running program (P, δ) to a new version (P', δ'). For this purpose, code and state transformations are performed [75, 76]. The program code is first updated and the actual program state δ then transformed to δ', so it is coherent with the new program code P' [75, 76]. The program state must be transformed into the representation P' expects. To this end, a state transformer is usually used.

Redundant hardware is often used as an alternative to dynamic software updating to modify a running system on the fly. In this approach, a secondary machine is employed. When a system update is desired, the new version of the code is loaded to the secondary system and the necessary state or information passed from the primary. After that, a role change between these machines is performed, where the primary machine is turned into the secondary, and the secondary into the primary one [75]. The program state might also be transferred.

Due to the availability of CPU and memory resources on modern computer systems and virtualization technologies, software-based replication instead of hardware ones are feasible, which are commonly extended to many machines, so availability is increased. For example, on IBM POWER processor-based servers, many firmware updates can be installed and activated without rebooting the system [77, 78]. Multiple firmware releases are supported in the field, so the current up-to-date firmware version can be used each time. However, changes to some server functions are not feasible during operation, such as initialization values for chip controls.

Another approach was followed by *P. Hosek* and *C. Cadar* [79, 80]. In this case, taking advantage of virtualization technologies, the new program version is executed in parallel with the old one, where their executions are synchronized. The aim of this technique is to maintain the stability of the old version while new features and bug fixes from the new version are also offered. When a divergence between the two versions occurs while running, the behaviour of the version which has not crashed is taken as the correct one. A prototype called MX was implemented, where several applications were executed on multi-version mode. The system is composed upon three main components. The first one is the Static Executable Analyser (SEA), which performs a static analysis on both version binaries. The second one is the Multi-eXecution Monitor (MXM), where both versions are executed concurrently. Finally, the Runtime Execution Manipulator (REM) selects between the available behaviours and resynchronizes both versions in case of divergence.

A dynamic software update process consists of three aspects, which are: **code transformation**, **state transformation** and the **update point**. The **code transformation** refers to the process of updating the executable code, while the **state transformation** stands for the procedure of trans-

forming the actual state of the program, so it is understandable by the new program. Finally, **update point** refers to the execution instance where the software update occurs. The term *update time* is also employed, which denotes the time instance when the update takes place. Both of them (**update point** and *update time*) refer to the same execution/time event. Figure 2.7 shows the timeline process of a DSU process.

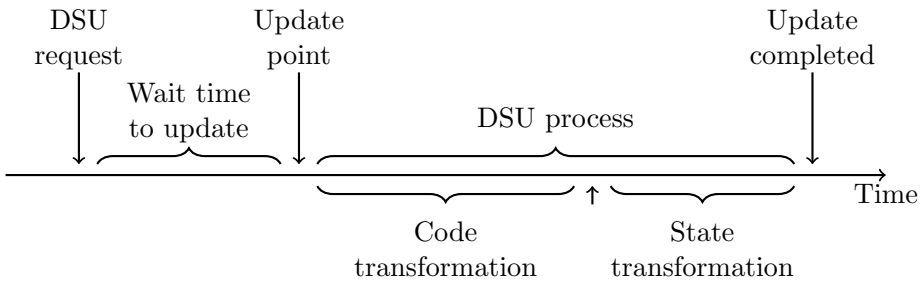


Figure 2.7: DSU process time-line

Code transformation

Code transformation refers to the process of updating, modifying or replacing the actual executable code into the new one. Figure 2.8 shows such procedure. In native or compiled languages such as C or C++, function-pointer indirections can be inserted by a source analyser, a compiler or manually by the developer. These indirections change the address of the function calls that will be next invoked. For this, a dummy jump at the start of each function is introduced as a trampoline. Note that, according to *Hayden et al.* [81], trampolines lead to a code injection attack susceptible computer program. Moreover, an extra level of indirection can be created, a modular approach, where function call and return indirections are managed from an indirection handling component or module. For this purpose a function table is designed, where every direct function call and returns are written. At update time, new updated function addresses are specified to the function table [82].

Another option is binary rewriting, where the whole program can be updated at once. This is achieved by accessing and modifying a specific region within the memory. Moreover, this technique, which is extremely platform dependent, permits inserting function indirection trampolines at runtime, in a similar way as buffer overflow attacks are carried out [83]. It might not be appropriate when an operating system with memory management tools is utilized. In this case, the DSU tools can not directly access or modify the

memory region or registers where the operating system kernel has placed the executable code or the program state data.

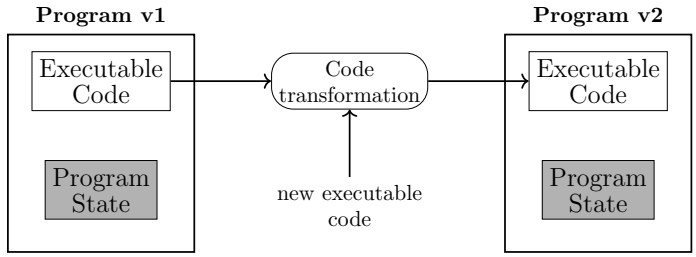


Figure 2.8: Code transformation procedure

Dynamic loading is commonly used to load the new code into the system, where the running program (usually the kernel) loads and accesses a new piece of code through an extension interface. This connection abstraction does not vary with time. This approach is regularly used by modern desktop operating system, such as in UNIX systems, to load and execute a given executable file which is stored within the file system [75]. A software architecture and a method for composable dynamic loading was proposed by *Shina et al.* [84] for time-critical system. If the program to be updated is executed within a virtual machine, the available infrastructure provided by the virtual machine can be used to load the new code, such as the Just-In-Time (JIT) compiler and the garbage collector [85].

Finally, the *unit of update*, as described by *Solarsky* [86], is the smallest software artefact which is possible to upgrade through a DSU process. While some DSU systems have been designed to upgrade a single software component, other ones are able to update a complete computer program. In case that it is not possible to upgrade at whole, an approach of dividing among several modules and update them progressively can be taken. The target application needs to be then built upon several modules.

State transformation

During an update, the program state must be transformed from the original representation to the new one. This procedure, which if depicted in Figure 2.9, is denoted the **state transformation** process. The current program state δ can include the stack frames, program counters, the heap and additional states maintained by the operating system such as open network connections. In order to adjust the current program state to the new execution code, a function which transforms a state object is used, which is referred to as a transformer function or state transformer. In some DSU systems, this transformation needs to be manually defined by the program-

mer, which is a laborious and error-prone task. In others, they try to synthesize the transformation operations. These two approaches can be also combined, where the DSU system first tries to automatically construct state transformer, whereas the supervision and/or adjustment from the developer is also required. In general, arbitrary transformations are possible, where no input from the programmer is necessary [75, 87].

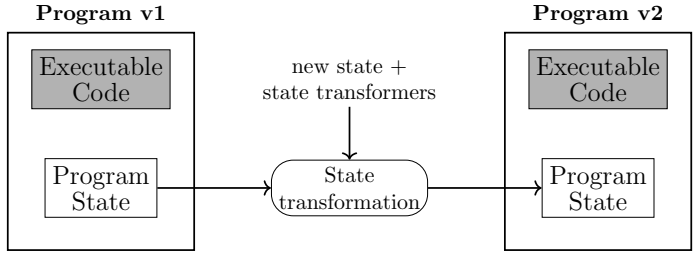


Figure 2.9: State transformation procedure

The state transformation process can be performed lazily or eagerly. Lazily means that a subset of the program state is transformed each time, as the related updated piece of the upgraded executable code is accessed, which causes steady-state overhead. As an advantage, stack-allocated values and global variables can be reached and handled easily. In contrast, all the state is updated at once if performed eagerly. While accomplishing in this mode, all other program executions are paused, so the state transformer functions are executed. This process wraps all the possible updating delays into the time instance where the state transformation is executed, while performing lazily, transformation costs are amortized [88].

Three main ways to update the current state data of the program exist. The first approach is to provide checkpointing and recovery features at the application level. These services would provide a method to serialize and deserialize a given program state, so it can be packed from an old program version and unpacked into an updated program version. This process is also known as state transferring or migration. C-strider is a type-aware heap transversal targeting C programs, which was used as the state transformation component for the Kitsune DSU system. Specifically, C-strider walks through the program heap. As the addresses located in this memory region are inspected, a small set callbacks are called. These invocations are used by the developer to build the program-independent serialization services [89, 90]. In this way, Ekiden is a state transfer updating library, which allows to pack the actual program state representation from the old running program and transfer it to the new one, where it will be unpacked and the program state re-instantiated [91]. A modification of such transferred program state is also possible.

The second way is to overwrite the old data *in place* with the new updated one at the same storage or memory [88]. However, the new state representation may require a bigger amount of memory than the old state representation. In order to address this issue *shadow data structure* and *type wrapping* techniques are used by the DSU systems [82]. Shadow data structures consist on adding extra fields which do not fit in the original data state. For this purpose, a pointer to a shadow structure is inserted at the end of each data structure. In contrast, in type wrapping , the DSU system holds extra unallocated memory while releasing the first program version. The aim of this activity is to keep free available space for future program state representations which may require larger memory than the previous ones. Nevertheless, the maximum size of the program data remains fixed.

Finally, the last option is to use indirections, referred as *struct replacement* by *de Pina* [82]. The new program state is moved to a new memory location, and represented as pointer of the underlying type. This technique deals with the problem of requiring extra memory by the new program state. Nevertheless, the memory management mechanism is needed, so the DSU system keeps track of available memory regions and it is able to release those memory regions where old unnecessary program data is located. These indirections inserted within the program should point out to the latest program data structure locations [88].

Update point

An update point, as depicted in Figure 2.10, is defined as the time instance and location where a program is updated. When an software updating request is received, the DSU mechanisms need to wait until an allowed update point is found. The DSU process starts then, where first the executable code is updated (**code transformation**) and the program state is then transformed (**state transformation**). The order of these steps may be switched. However, it shall be ensured that the running executable code uses the correct representation of the state.

In order to maintain the correctness of the running program, dynamic updates shall not take place while affected data or executable code is employed. The program might crash. Consequently, it is crucial to properly define when a software update can be safely realized, which is indicated by the **update point**. As explained by Hayden [92], three strategies exist to determine the update point of a running program: Activeness Safety (AS), Con-Freeness Safety (CFS) and Manual. A static analysis is typically performed to evaluate if the selected update point satisfies safety guarantees [87, 93].

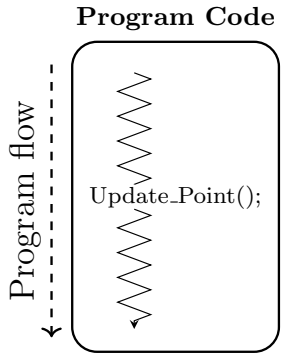


Figure 2.10: Update point specification

On the one hand, in the AS approach, a software update may occur only if the piece of executable code which will be upgraded is not active. Therefore, no updated function should reside on the call stack at update time. Safety is guaranteed with the confirmation that the old code would not be accessible anymore. The software updated is delayed until a quiescence state is reached. This approach is most likely the most widespread one used by the commercial DSU systems. On the other hand, CFS allows updates to active executable program if the old program that will be executed after the upgrade will never access the program state or a call a function whose type signature has been modified. This approach was proposed by Stoyle [94], after he observed that AS can be too restrictive in some applications.

Lastly, in the manual approach, it is up to the developer to decide in which point may the executing program be updated. In this case, it is the responsibility of the developer to specify those update points. This annotation is commonly inserted in long-running loops, where no resources are held. In case active calls are encountered on the stack, a stack reconstruction procedure can be applied. This method was employed in Upstare. Software upgrade safety depends then on the maintainer/developer judgement or on the adopted stack reconstruction algorithm and implementation.

In case that the application is multi-threaded, the DSU of such application becomes more challenging. Firstly, the executable code and the program state from each thread need to be transformed. Secondly, it is required that all threats reach an update point and wait to the others so the DSU process can begin. Nevertheless, the possibility of unbounded delays of those threads when reaching an update point could lead to a service interruption from the application. Furthermore, a deadlock may occur, when a given thread is waiting for accessing given resource before reaching an update point while another one has already reached it without releasing the resource. The application would then be halted [76, 92].

In order to be feasible for multi-threaded program to be updated, the following three preconditions are needed to be met [92], which are:

1. An update point shall be repeatedly reached by every long-running thread.
2. A thread shall not hold resources.
3. The program to be updated should not matter the order in which threads are restarted in the new version.

The DSU procedure for a multi-threaded program starts with all threads reaching an update point. At this point, the main execution thread updates all the threads as if they were ordinary single-threaded applications. The main thread continues executing until an update point is reached, where it updates itself. Finally, each of the threads are relaunched then, initializing or migrating first the new program state data [92].

3 State of the Art

In this chapter, the state of the art is presented. On the one hand, a review of industrial safety and security standards with respect to software updates is provided. On the other hand, an analysis of existing dynamic software updating techniques is given. In this review, the requirements for dynamic software updates in safe and secure systems are also collected.

3.1 Standards

The IEC 61508 is the main international standard for electrical, electronic and programmable electronic safety-related systems, which is contemplated as the fundamental functional safety standard [13]. The requirements for ensuring that systems are designed, implemented, operated and maintained to provide the required safety integrity level (SIL) are specified. Although the IEC 61508 standard is generic and applicable to all kinds of industry, it has been adapted to application-specific safety domains. Table 3.1 shows some of the domain specific safety standards.

Domain	**Standard**	**Name**
Automotive	ISO 26262 [95]	*Road vehicles – Functional safety*
Railway	IEC 62278 [96]	*Specification and demonstration of reliability, availability, maintainability and safety*
	IEC 62279 [97]	*Communication, signalling and processing systems - Software for railway control and protection systems*
	IEC 62425 [98]	*Communication, signalling and processing systems - Safety-related electronic systems for signalling*
Process industry	IEC 61511 [99]	*Functional safety - Safety instrumented systems for the process industry sector*
Machinery	IEC 62061 [100]	*Safety of machinery - Functional safety of safety-related electrical, electronic and programmable electronic control systems*

Table 3.1: Domain specific safety standards

An state of the art about safety and security standards was realized at the ITEA3 MERgE project, where multi-concerns, particularly focused on safety and security co-engineering were tried to be efficiently handled [4, 5, 56]. Figure 3.1 presents the analysed safety and security standards, which are depicted as undashed boxes. The diagram also shows related standards which may be taken into account for the development of safe and secure systems, even if safety or security concerns are not directly addressed on them. These standards, shown as dashed elements, are not studied.

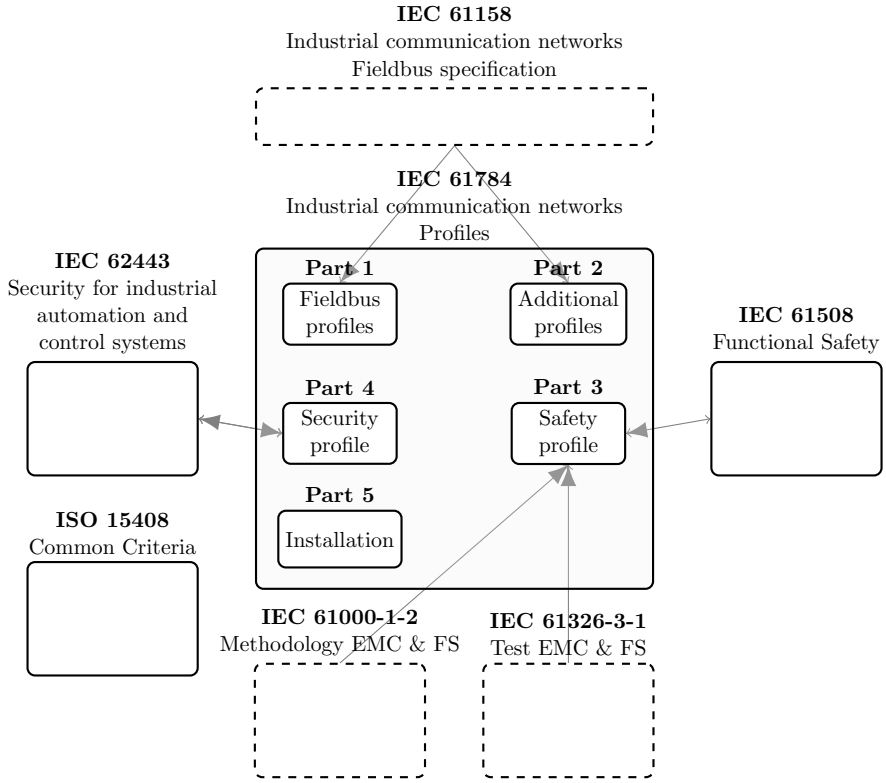


Figure 3.1: Investigated safety, security and related standards

As pointed out by [5], it seems that each safety standardisation institution produces its own domain-specific security regulation instead of creating generic ones for security. In Figure 3.1, it can be seen that the IEC 61784 standard is used to associate safety and security domain standards, which is also illustrated within the mentioned normative [101]. Even International Organization for Standardization (ISO) 15408 is reviewed, there is not a direct and clear association with other norms. This is the reason why this element is not connected to any other item in the diagram.

As far as security is concerned, several well-known standards are widely applied on the development and certification and management of IT system and devices, such as the ISO 27000 series or the Common Criteria. The ISO 27000 series provide information security management and best practice recommendations [102]. This normative introduces the Information Security Management System (ISMS), whose purpose is to manage information security risks through information security controls. In addition, continuous feedback and improving activities are incorporated to the ISMS. The aim of these adjustments is to respond to new threats and/or vulnerabilities.

The ISO 27000 framework is applicable to any kind of organization, which covers more than just cyber-security concerns. Organizations are encouraged to assess, supervise and handle their information security risks by following the guidances and recommendations defined in the norm. The standard is reviewed and updated more or less every five years. It is expected that its next version will cover cyber-security and digital forensics aspects. On the contrary, the Common Criteria, also known as ISO 15408 [103], is a framework where the functional and assurance requirements for a given product are first defined and successively evaluated by a security evaluation laboratory to determine if they actually satisfy those claims. Even it is mainly focused for IT environments, industrial control systems can also be evaluated.

In order to address security issues on IACSs, the International Society of Automation (ISA) created the ISA99 standard, which addresses the security of industrial automation and control systems. These technical documents were then renumbered to the corresponding IEC standards, the IEC 62443 series.

3.1.1 IEC 62443

The ISA/IEC 62443 is a series of standards, technical reports, and related information that define procedures for implementing electronically secure IACS [14]. This standard, created by the International Society of Automation ISA, was originally named ISA 99. Nevertheless, this normative was renumbered to ISA 62443 in 2010. The purpose of this modification was to align ISA documents with the analogous IEC standards. The ISA-62443 standards and technical reports are organized into four general categories, which are *General*, *Policies and Procedures*, *System* and *Component*.

Table 3.2 provides a short description for each general category.

Name	Part	Description
General	IEC 62443-1-X	Provides background information such as concepts, terminology and metrics
Policies and procedures	IEC 62443-2-X	Defines the necessary elements to establish a cyber-security management system, security policies, practices and patch management processes
System	IEC 62443-3-X	Provides system development requirements and guidances
Component	IEC 62443-4-X	Provides product development and technical requirements, intended for product vendors

Table 3.2: General categories of the IEC 62443 standard

Within the scope of IEC 62443, special attention is given to software updates. An evidence of it is that it provides a dedicated document deferring to patch management within the **Policies and procedures** category. This technical report is the **IEC 62443-2-3: Patch management in the IACS environment**, which states that patch management is an element of a complete cyber security-strategy, where cyber-security vulnerabilities, bugs, operability and reliability issues are resolved [15]. Nevertheless, the standard does not differentiate among operative system, library or application-oriented patches. The aim of it is to provide a generic guidance for all type of patches. Note that, as stated in this technical document, "Applying patches is a risk management decision". The software upgrade may be rejected or delayed if the cost to apply the patch is greater than the risk evaluated cost [15]. Patch management is defined by the IEC 62443-2-3 [15] "set of processes used to monitor patch releases, decide which patches should be installed to which system under consideration, if the patch should be tested prior to installation on a production system under consideration, at which specified time the patch should be installed and of tracking the successful installation".

At some point, because of the long operational periods of IACSs, a given product may become obsolete and/or could be no longer supported by the product supplier. However, new security vulnerabilities might still be discovered on such products. Asset owner should then adopt other mitigation strategies when software updates on the target system are not an option.

Patch lifecycle

The patch lifecycle defines a series of states through which a patch passes from the time that it is available by a third party or a product supplier until it is installed or rejected by the asset owner. Figure 3.2, which is a

replica of the patch state model scheme provided in the IEC 62443-2-3 [15] technical document, depicts the patch lifecycle state model.

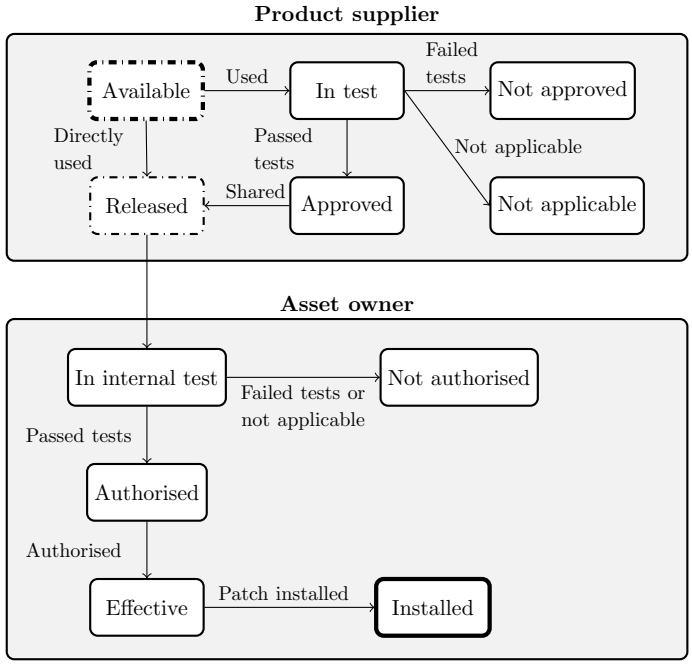


Figure 3.2: Patch state model

Not all available patches will be approved and installed. Thus, it is important to keep the track of all available patches for an efficient patch management procedure. In addition, clear evidences will be needed to gather to ensure that the system will behave correctly functional once the patch is applied. As it can be observed in Figure 3.2, the patch lifecycle state model is divided into two parts. The first part corresponds to the states maintained by the product supplier. In contrast, the second part conforms to the states associated with the asset owner. Furthermore, as illustrated by the standard, the asset owner is also able to directly gather available patches and release, share or distribute them. This is the reason why the *Available* and *Released* states are shown as dashed elements in Figure 3.2.

Table 3.3 gives the description of each of the states defined in the patch lifecyle model [15] and shown in Figure 3.2.

Patch state	Patch state definition
Available	The patch has been provided by a third party or an IACS supplier but has not been tested
In test	The patch is being tested
Not approved	The patch has failed the testing and should not be used, unless and until the patch has been *Approved*
Not applicable	The patch has been tested and is not considered relevant to IACS use
Approved	The patch has passed testing
Released	The patch is released for use or third party, or the patch may be directly applicable by the asset owner for their internally developed systems
In internal test	The patch is being tested by the testing team
Not authorised	The patch has failed internal testing, or may not be applicable
Authorised	The patch is released and meets company standards for updatable devices, or by inspection did not need testing
Effective	The patch is posted for use
Installed	The patch is installed on the system

Table 3.3: Patch lifecycle states

Guidance on patching

Two different guides on patching are provided by the IEC 62443-2-3 technical document [15]. The first one is oriented to product suppliers, while the second one instead asset owners. The *product supplier guidance on patching* provides a reference procedure to develop and distribute new software updates. This guidance defines four major activities:

- Discovery of vulnerabilities: Guidance identifying new cyber-security vulnerabilities, and monitoring third-party software products which are dependant.
- Development of security updates: Guidance developing and validating updates, where all the dependent third-party software has to be also taken into account.
- Distribution of security information: Guidance distributing patches and associated information, which shall be proceeded securely and on time.
- Communication and outreach: Guidance about how the product supplier should maintain communication with asset owners.

On the contrary, the main goal of *asset owner guidance on patching* is to describe patch management procedures and processes for the asset owner. Four major activities for patch management are explained. These activities are: *Information gathering*, *Project planning and implementation*, *Procedures and policies for patch management* and *Operating a patch management system*. Once the procedure for patching is defined and documented, it can be shared with the individuals who are responsible of executing it. This task is essential, so patching activities are carried out efficiently and appropriately by the personal. This assignment is applicable for both product suppliers and asset owners.

Patch information

"Determining the compatibility of patches can be a complex task" [15]. Before releasing the new software update, several tests are performed by the product supplier to figure out if the patch fits on their industrial control system products. Moreover, it may be required that the industrial control product is accurately tested and verified on a experimentation environment with the patch applied to it prior to the installation within the real production system.

From the users perspective, it is important to keep track of the different patches concerning industrial control systems. Consequently, a patch management system shall be implemented, which should content knowledge about:

- What patches are available
- The applicability of those available patches
- Previous testing results on the installed products
- Recommendations from the product supplier

Industrial control system users operate products from separate suppliers. Thus, patch compatibility information for those systems needs to be managed. This information determines if a specific patch for a given product from a given supplier is compatible with a third party product from another vendor. It also points out which versions of the software have been tested and verified. This information aids the users to decide which patch should be applied or not. Nevertheless, it may be a difficult task to handle all the patch compatibility information due to the lack of an unified approach to exchange patch compatibility information. Thus, in order to share the minimal compatibility information of a patch, a Vendor Patch Compatibility (VPC) file format is proposed within the technical report IEC 62443-2-3, which is based on the eXtensible Markup Language (XML) specifications.

More detailed definition of this information exchange format can be found at the Annex A [15].

3.1.2 ISO 15408

The ISO 15408, also known as Common Criteria, is a framework in which users can indicate their security functional and assurance requirements (SFR and SAR respectively) through Protection Profiles (PP). Manufacturers can then develop their products upon the specifications described in those protection profiles and make claims about security attributes of their products. A testing laboratory would be the responsible of evaluating those products to determine if they meet the claims manifested by the vendors, where the level of confidence is also established [103, 72, 74].

A certification process starts with a *Security Target* (ST) document. This report identifies the security properties of the product or system which is intended to be evaluated, referred as Target Of Evaluation (TOE). It includes an overview of the product or system, possible security threats, detailed information on the implementation of all security features and any claims against a protection profile, which defines security requirements for a class of security device such as network firewalls.

Even the Common Criteria is mainly focused on IT environments, a protection profile for industrial systems exits, which is the *System Protection Profile for Industrial Control Systems* (SPP-ICS) provided by the NIST [104]. This protection profile includes SFRs and SARs. Nevertheless, as stated within the document, it has been written in such a way that it may be used as the basis for preparing a System Security Target for a specific IACS or as the basis for a more detailed security PP. Although a new protection profile for safety-related communications in railway was proposed by [60], at the time of writing protection profiles targeting industrial control and/or safety-critical systems are missing. Thus, albeit this standard can be used to certify wide range of IT security products such as operative systems, databases or smart cards, protection profiles for the industrial domain are needed to be defined still.

Some protection profiles introduce objectives and requirements concerning software or firmware updates. An example of such profiles is the Protection Profile for *Smart Card Reader with PIN-Pad supporting eID based on Extended Access control*, created by the *Bundesamt für Sicherheit in der Informationstechnik*[105]. As defined in this document, software updates must be signed, and installed if the signature is correctly verified. On the contrary, industrial control system oriented protection profiles do not include any requirement concerning software updates. Indeed, within the Protection Profile for safety-related communication in railway automation,

objectives, requirements or procedures related to software updates were not contemplated [60].

Finally, the security product or system are rated according to the Evaluation Assurance Level (EAL), which is a numerical rating which indicates the depth and rigour of the evaluation. Seven levels are defined, from EAL1 being the lowest one, up to EAL7. Note that, a higher level does not mean that the product or the system has more and/or better security capabilities, but the level of confidence against PPs.

3.1.3 IEC 61784

The *digital data communications for measurement and control* IEC 61784 standard defines a set of protocols specific communication profiles based mainly on the IEC 61158 norm [101]. The purpose of the IEC 61784 is to aid to properly state the compliance to the IEC 61158 series, where fieldbuses for industrial control systems are specified. Functional safety and information security profiles are also addressed. As stated in the third part, additional security requirements are detailed in IEC 62443 series. The IEC 61784 standard includes several communication profile families, which specify one or more communication profiles. The specifications for the communication protocol stack are then determined for each profile, where the minimal set of required services at the *Application* layer are provided. In absence of this layer, minimal set of required services at the *Data Link* layer are indicated. Specification of options at the intermediate layers are also defined. It has to be mentioned that devices or systems complying to the same communication profile will accomplish a direct interoperability between them. The IEC-61784 standard is divided among 5 parts: *Profile sets for continuous and discrete manufacturing relative to fieldbus use in industrial control systems* (IEC 61784-1), *Physical layer specification and service definition* (IEC 61784-2), *Functional safety fieldbuses* (IEC 61784-3), *Profiles for secure communications in industrial network* (IEC 61784-4) and *Installation* (IEC 61784-5).

The IEC-61784-3 document describes important principles for functional safety communications, and specifies certain safety communication layers based on the communication profiles and protocol layers defined in the first and second parts of the IEC 61784 and in the IEC 61158 standards. These specifications, which are intended to be applied employing fieldbus technology within a distributed network, are aligned with the requirements of the IEC 61508 functional safety norms [13]. Safety communications provide a mandatory confidence in the information transportation between two or more participants in a safety-related systems and enough reliance of safe behaviour in the event of communication errors or failures. Analogously

specified in the IEC 61508 standard, a SIL level is defined for the safety-related information transportation channel, depending on the functional safety communication profile that has been applied.

As far as profiles for secure communications in industrial network is concerned, this part of the IEC 61784 standard provides security communication profiles [101]. Similar to functional safety information transmissions, communication profiles and protocol layers defined by the IEC 61158 and other IEC 61784 parts are redefined and improved from the security point of view. Some profiles permit to be used as a black channel for functional safety applications. For this purpose, the delay of the secured channel are bounded. Different network application scenarios are contemplated by those security communication profiles, where different security mechanisms are supported in each one. Twelve possible scenarios are considered, which they vary from a *standalone embedded device* until an *external network interconnection to a control network* or an *interactive remote access to a control network*. However, it should be noted that just a draft version from 2005 of IEC 61784-4 has been found. The contents of it have been overlapped by the IEC 62443 standard.

3.1.4 IEC 61508

The IEC 61508 normative is contemplated as the basic functional safety standard applicable to all domains. It covers the safety issues of electrical, electronic or programmable electronic systems or devices, but concerns like long-term exposure to a toxic substance or an electrical shock are not within the scope of it [13, 44]. It is divided among seven parts, where the first three parts contain the normative itself and the rest ones are guidelines and examples. The normative aims at reducing the risk, which is a function of frequency or probability of the hazardous event and the event consequence severity, to a tolerable level. For this purpose, safety functions are applied.

The standard covers the complete safety life cycle for the development of the system, consisted on sixteen main steps. They are divided among three groups: *analysis*, *realization* and *operation*. These phases specify how should be developed and maintained a safety-related system (both hardware and software). At the hazard and risk analysis phase, the hazards, hazardous events and hazardous situations are agreed so the risks associated with those events can be determined. Due to the security threats, this evaluation needs to be extended in case malevolent or unauthorised actions are identified. During this security threat analysis, deliberate misuse, vandalism and criminality are taken into account [44].

However, even if safety hazards and risk will not change through operational period, security threats evolve continuously. This means that security

issues need to be addressed at the *overall operation, maintenance and repair* phase, where it shall be ensured that the functional safety is maintained to the specified level. An example of such scenario would be the modification of the communication stack software when a new security-enhanced communication protocol is requested to be adopted. Thus, a security patch management system may be needed as required by the IEC 62443 standard.

Four different safety integrity levels are established. Each level defines the risks involved in the system applications, where SIL4 is used for applications entailing high risks. These measures are taken from the first part of the standard. For systems that operate on a low demand mode, SIL specifies an allowable probability that the system will fail to respond on demand. These metrics are depicted in the following table 3.4.

Safety integrity level	**Average probability of dangerous failure on demand** (PFD_{avg})
SIL1	$\geq 10^{-2}$ to $< 10^{-1}$
SIL2	$\geq 10^{-3}$ to $< 10^{-2}$
SIL3	$\geq 10^{-4}$ to $< 10^{-3}$
SIL4	$\geq 10^{-5}$ to $< 10^{-4}$

Table 3.4: Safety integrity levels - target failure measures for a safety function operating in low demand mode of operation

For systems that operate on continuous mode or systems that operate on high demand mode, SIL specifies an allowable frequency of dangerous failure. These metrics are depicted in the following table 3.5.

Safety Integrity Level	**Average frecuency of a dangerous failure per hour** (PFH)
SIL1	$\geq 10^{-6}$ to $< 10^{-5}$
SIL2	$\geq 10^{-7}$ to $< 10^{-6}$
SIL3	$\geq 10^{-8}$ to $< 10^{-7}$
SIL4	$\geq 10^{-9}$ to $< 10^{-8}$

Table 3.5: Safety integrity levels - target failure measures for a safety function operating in high demand mode of operation or continuous mode of operation

3.1.5 Evaluation

The IEC 61508 [13, 44] and the IEC 62443 [14] could be considered the reference standards when it comes to the design, development and maintenance of safe and secure systems. In addition, the IEC 61784 [101] could be also employed in case of safe and secure communications are required. Since the IEC 62443 addresses security issues and challenges for generic industrial control systems, it could also be applied as the security-related guide for domain specific safety standards, such as automotive, railway or process industry [95, 96, 97, 98, 99].

At the time of writing, the Common Criteria framework [103, 72, 74, 104] provides the necessary facilities for the design, development and maintenance of secure industrial systems. However, protection profiles for industrial automation systems, which shall also define objectives and requirements with respect to software updates, are missing. In case of the ISO 27000 series [102], even though information security management and best practice recommendations are given, it does not provide any guideline, procedure and/or requirements for the design and development of secure systems. Concepts and methods from the ISMS, and the patch management guidelines proposed by the IEC 62443-2-3 [15] technical document could be merged or combined with the aim of building an effective and complete patch management system for safe and secure industrial systems.

3.2 Analysis of Existing DSU Systems

Over the years, many DSU systems have been proposed. These systems target many kinds of applications, from real-time control purposes to servers and databases [106, 107]. Moreover, formal approaches have also been proposed, where DSU support at the programming language level is investigated. An underlying theory for languages which offer DSU features is presented by Stoyle [94], named Proteus. This theory is a program calculus and it was applied to the design and implementation of updatable C-programs. The most important challenges reside on how to address the unsafe features of C programming language, and how to design an efficient DSU feature.

Some actual programming languages already provide DSU support. Four programming languages were analysed by *de Pina* [82], which are: Common Lisp, Smalltalk, Erlang and UpgradeJ. Even that these programming languages provide high-level DSU features, the developer is required to write the target applications on those languages. Consequently, the usage of these approaches is restricted for those applications developed from scratch in these languages. Incompatibility issues while integrating already existing libraries with the target application written in such programming language may also arise.

A review of techniques, evaluation metrics and a survey of existing DSU systems have been provided [106, 107]. These systems are then categorized according to the used or characterized **code transformation**, **state transformation** and **update point** techniques and attributes. The DSU mechanisms are also assessed and discussed against the defined evaluation metrics. In this subsection, DSU systems which could be feasible for industrial control applications are analysed. They have been divided into three categories, as similarly done by *de Pina* [82] and *Seifzadeh et al.* [106], which are:

- **Compiled application**-oriented DSU systems which target applications compiled to an executable binary. These objects are then natively executed.
- **Kernel**-oriented DSU systems which target the core of an operating system.
- **Real-time**-oriented DSU systems, which have specifically been created for real-time, embedded system or IACSs.

Compiled application and **Kernel**-oriented DSU systems could be used or be adapted to embedded system, real-time or IACSs, while **Compiled application**-oriented ones could be reused for **Kernel** ones.

Table 3.6 presents the investigated DSU systems for each of the categories. In total, twenty systems have been analysed and compared.

Name	Target	Date
DLpop	Compiled Application	2001
OPUS		2005
DynSec		2013
POLUS		2007
UpStare		2009
Ginseng		2008
Ekiden		2011
Kitsune		2012
LUCOS	Kernel	2006
DynAMOS		2007
KSplice		2009
K42		2006
PROTEOS		2013
DURTS	Real-time	2004
EmbedDSU		2011
Gracioli		2014
EcoDSU		2008
Seif-Real		2009
Wahler		2009
FASA		2014

Table 3.6: Analysed DSU systems

Note that the DSU systems are not ordered according to the date within each category. A similar arrangement used by *de Pina* [82] and *Seifzadeh et al.* [106] has been employed. Besides, DSU systems not suitable for industrial applications have been omitted and new ones added. After the analysis of each investigated DSU system, a classification is provided, where the different properties and capabilities are compared and summarized.

Compiled application

Compiled application refers to a software application which is compiled into an executable binary. This binary code is then executed natively on the target. The C programming language is probably the most common language for the development of compiled applications. As pointed out by *de Pina* [82], this programming language is also used for the development of

operating system kernels. Thus, DSU systems targeting operating system kernels were included in this group. Nevertheless, in this investigation, these DSU systems have been classified into their own category.

DLpop

DLpop is a dynamic linking framework suitable for DSU, which targets UNIX execution environments [75]. The Typed Assembly Language (TAL) and Popcorn (type-safe C) languages were used with the aim of providing safety and robustness. This DSU system provides TAL/Load, which is a type-safe version of DLopen [108], a dynamic-linking procedure for C on UNIX systems. The DSU approach consists of four different phases, which is applied for each software module that is wanted to be dynamically updated. The steps are:

1. Generate a dynamic patch.
2. Compile the dynamic patch to a loadable TAL file.
3. Link and update dynamically the file on the running program.
4. Switch to the patch from the running program.

First of all, a dynamic patch needs to be generated, which is defined as the difference between two versions of a software module. In contrast to static patch a dynamic patch contains both the updated program code plus the additional code and data required to switch to the new program version. This information may contain changes to code, program data and type definitions. Also, stub functions are provided where call indirections are specified. Consequently, a dynamic patch is defined as tuple of $(f, S, stubset)$, where f is the new program code, S is the state transformer which may modify the static data and the heap, and *stubset* is a set of mappings from program functions to their corresponding stubs. These dynamic patches are constructed by a dynamic patch building system.

A patch is described by a patch description file containing four parts: the implementation filename, the interface code filename, the shared type definitions, and the type definitions to rename. The first two parts describe the patch: its implementation in the first file, and the state transformer and stub functions in the second file. The final two parts are for type namespace bookkeeping. The shared type definitions are those types that the new file has in common with the old one, while the changed definitions are in the renaming list, along with a new name to use for each. The compiler uses this information to syntactically replace occurrences of the old name with the new one.

In order to be loadable, the dynamic patch it has to be compiled and linked along with the updating library. At this stage, a Global Offset Table (GOT) is created in each binary, where all references to data are inserted and a hash table for each binary file is included. These tables are linked and unlinked from the GOT when program modules are loaded and unloaded. In addition, code, data, and type definitions may be modified. Finally, DLpop is used to enable the dynamic updating. This utility allows to first load dynamically a given dynamic patch and build a hash table for symbols. The update module is then linked and the rest of the program re-linked.

OPUS

The Online Patches and Updates for Security (OPUS) is a DSU system which works transparently with the classic building tools found on UNIX systems. It consists on three main steps, which are: patch analysis, patch generation and patch application [93]. Initially, a static patch is performed, so the changes between the source files are inspected and reported. This information, which will be later on passed to the compiler, is used to decide if the static patch requires a static analysis or not.

At the static analysis phase, the compatibility and safety attributes of the patch are investigated. In case that the static patch does not meet the approval conditions, the patch is rejected, providing an error message to the developer. This is assessed by evaluating the previous static patch information. On the contrary, the static patch is further analysed conservatively in order to ensure that, once dynamically applied, it will lead to an unsafe DSU case. After the patch is validated and it is free of errors and, desirably, also of warnings, the patch generation step starts. Firstly, files and the information required to build dynamic patches are gathered, where changes to global variables and executable functions are identified. This information is then extracted and explicit binary code created.

A dynamic patch object is built next, where all these binary objects are packed together with patch definition information. This is then transferred to the target. At this point, a patch installer service, which is able to examine and modify the application process address, is dispatched. This service blocks the execution of the old program, loads the new executable code (using the *dlopen* and *dlsym* utilities) and applies the dynamic patch by redirecting calls. For this, the first instruction of the old code is overwritten. Nevertheless, AS is verified before applying the dynamic patch. A similar interface to usual C compilers is provided by OPUS, where errors and warnings from the source analysis, compilation and dynamic patch generation steps are presented. These processes are invoked by the developer. This DSU system supports the upgrade of multi-threading applications.

DynSec

DynSec is a DSU system, which is built on top of a Dynamic Binary Translator (DBT). Once the new binary file is loaded, this code is rewritten through with the DBT. A sandbox virtualization layer is then employed, where apart from the DSU utilities, the protection of the system against software upgrades leading to unsafe situations is also provided. System call policy, stack integrity and code integrity are safeguarded [109].

In DynSec, a DSU service is created, which is responsible of processing the requests gotten from the developer. This utility, which is executed outside the sandbox, manages the DSU process. This procedure has three main steps:

1. The patching service waits for a DSU request.
2. The patching service synchronizes all running application threads, so they can reach a safe update point.
3. A DSU is applied for each application thread. For this purpose, their code cache is flushed.

In this DSU system, a dynamic patch is a binary file, where after specifying the number of patched instructions, the information about these instructions is given. This data contains the address of the instruction, previous and actual length of the instruction and the new native code. DynSec examines which libraries, modules and loaded symbols are charged and utilized. In addition, it keeps tracks of all running threads by monitoring system calls, so it can be ensured that all the application threads reach a safe update point. Once the patch is applied to those instructions specified by the developer using the DBT, these patched instructions are executed within the sandbox (virtualization layer).

POLUS

POwerful Live Updating System (POLUS) is a software maintenance utility which permits to dynamically update the new application software code and data [110]. This DSU system permits the coexistence of two different data representations. State synchronization functions are invoked when a write instruction is called, so the coherence of both data versions are maintained. The DSU is safely completed when no more access to the old version data exists. Support for both single-threaded and multi-threaded applications is provided.

POLUS includes a patch constructor, a patch injector and the runtime library. The patch constructor is a source-to-source compiler, which is able to recognise the syntactic and semantics differences between two program

versions. In contrast, the patch injector is the process executing in the target which is capable of applying a dynamic patch. Finally, the runtime library provide DSU-oriented tools for the dynamic patch management. These userspace utilities are linked to the patch injector, so the developer can send requests to the POLUS core, such as a DSU command.

First of all a static patch is produced. This data is then used to generate dynamic patch by the patch constructor. In POLUS, a dynamic patch is a whole-program patch compressed on a single file. This file contains the modified code, changed global variables, changed type definitions and additional supportive code to maintain the consistency between new functions and variables, including state transformer functions. The created dynamic patch can be applied by invoking the patch injector. For this, the developer needs to use the runtime library tools. The patch injector will insert an indirect jump instruction just at the header of the old executable block, so function calls are redirected to the new updated ones.

As far as the state transformation is concerned, POLUS is not able to update function level state information, such as local variables or local stacks. Only global visible state is visible and considered. In POLUS the old and the new program state representations can coexist concurrently. In order to avoid inconsistencies and concurrent access between those state representations, state synchronization functions are used. These methods are invoked by the signal handler, so data from one program state version is updated to the other one. The DSU procedure is completed when all threads accessing old data are not active anymore.

UpStare

UpStare is collection of tools which aims at providing DSU capabilities. The framework comprises a patch generator, a custom compiler, a DSU runtime environment and a DSU tool [76, 111, 112]. The target application, which could be a multi-threaded program, is required to be compiled using the custom compiler. Nevertheless, it is a straightforward process. The call to an existing compiler in the application building process has to be changed, so instead of calling for example, the *gcc* compiler, the UpStare one shall be invoked. There is no need to modify the source code [112].

When a new software version is released, the source code of it in addition to the old one, is provided to the patch generator. The source code of a dynamic patch is then created, where information about new functions, new global variables and modified data type definitions for both global or declared on the stack are incorporated. A partial state mapping and data transformers are also produced. Typically, the user is required to adjust the state mapping functions and to determine the update points which are

inserted at the next stage. This dynamic patch source code is then compiled using the UpStare compiler, which is based on the Common Intermediate Language (CIL) framework [113], so an executable update patch is generated. The executable dynamic patch shall be saved on the target machine. When DSU of a given application is desired, the developer should send an update request to the runtime environment. This is achieved through a TCP/IP connection, which means that a software update across the network is possible. The runtime environment is statically linked to the target program and it is able to dynamically load the dynamic patch using the *dlopen* function. Once the update request is received, the runtime environment blocks all the application threads and starts the stack reconstruction procedure.

The stack reconstruction is the essence of this DSU system, which is automatically implemented by the UpStare compiler. For this purpose, a source-to-source transformation is applied. A program is placed in stack reconstruction mode when after receiving the update request, all threads are blocked. At this point two main steps are performed. First of all, the program stack frames of every thread to be updated are automatically saved and unrolled. State transformer functions are then called by the runtime environment to convert the old state representation to the new one. Next, the stack of each thread needs to be reconstructed. At this point, the formal parameters, the execution point and local variables of a function are restored. This is recursively reproduced until the whole stack is rebuilt. Finally, after the stack reconstruction process, the update is complete and the runtime environment starts to resume the application threads.

However, library functions and signal handlers are not compatible with this technique. This information is stored within the operating system, so UpStare avoids resetting the signal handlers. Instead, function pointer indirection is used to initiate calls. If the application, which is wanted to be updated dynamically, is executing a signal handler at the time of receiving the update request, the software upgrade is rejected.

Ginseng

Ginseng is a tool suite for building updatable programs, so they can be updated during runtime [114]. Before designing the tool suite, several long running C program evaluations were studied. For this, the *ASTdiff* tool was developed, which is able to parse and compare the two source code versions of the program. Six server applications were dynamically updated with Ginseng, three single-threaded and three multi-threaded [114]. It is composed upon a patch generator, a compiler and a runtime system, where, for the first two components, the CIL framework [113] was used.

The patch generator compares two versions of a program, where changes at the global variables, functions and types are reported. Based on this information, a type transformer is automatically generated, so the old type representations are mapped into the new ones, where adjustments from the developer are possible. In contrast, the state transformation function may be written by the developer. This function is invoked at the update time and it defines how the actual program state representation should be adapted so it is consistent with the new executable program. All this data is then sent to the compiler.

Two tasks are accomplished by the Ginseng compiler. On the one hand, target applications are compiled and the dynamic patch generated. Safety analysis are performed to ensure that safety properties will not be disrupted while DSU tasks are carried out. State transformations are also specified, so the new program data is aligned with the new executable code, as well as type transformers. These conversion methods have been created at the patch generation step. Type wrappers are used, where new types with larger amount of memory are defined as the replacement for the old ones. In case of an update which requires more memory, previously defined types will hold usable memory area. Moreover, function indirections are inserted. A level of indirection is included through a global variable, so the new updated function can be invoked from the old code. On the other hand, static analyses are performed on the program source code. The aim of this study is to guarantee that the DSU process will not lead to unsafe execution incidents.

The Ginseng runtime system is responsible of executing the DSU process. When a DSU command is received, the runtime system waits until the program reaches an update point, where the dynamic patch is loaded and linked. These update points are automatically introduced by the compiler. Nevertheless, they can be added or modified by the developer.

Ekiden

Ekiden is a state transfer updating library, which allows to pack the actual program state representation from the old running program and transfer it to the new one, where it will be unpacked and the program state re-instantiated [91]. At this point, the new program can be launched and the old one halted. It offers several advantages. First, there is no need of static analyses, and compiler optimizations can be enabled when the new program version is compiled. Secondly, only the state which changes through time is required to be transmitted and upgraded because the new state is automatically initialized. However, multiple instances of the program need to simultaneously to executed to accomplish the transfer process.

The source code of the target application needs to be modified in order support Ekiden. The program state to be transferred and updated later on needs to be defined as the tagged program state. Apart from that, update points need to be manually specified by the developer. The author recommends inserting those update points at the starting point of each long-running loop [91]. Ekiden is used to serialize the program state, and de-serializing it into the new program version. A checkpointing approach is adopted, where a tool parses the annotations written by the developer and creates the serialized object. Ekiden is the precursor of the Kitsune DSU. As stated by *Hayden* [92], who is the author of both schemes (Ekiden and Kitsune), the state transferring cost was notable in Ekiden. For this reason, a new strategy was chosen in Kitsune, so in-place data and code updates are possible.

Kitsune

Kitsune, which was designed after the weaknesses of Ekiden were found, provides a program level DSU system [92, 81]. In the same manner as other DSU systems, **code transformation**, **state transformation** and **update point** attributes are needed to be specified to the original program source. Whole program are upgraded by this DSU system, which is able to update also multi-threaded programs. For this, each thread is updated stopped and updated separately. Once completed, the Kitsune runtime system re-launches each upgraded thread, where first of all, data initialization and migration tasks are performed.

A state transformation tool is provided, which is named *xfgen*. This tool is used to specify how the global state of the old program version will be transformed and migrated to the new program version. This information is then passed to the compiler. For this, the data contained in the *xf* file, which is generated by the *xfgen* tool, is converted to C code. Kitsune uses standard compilation utilities, where the updatable program is compiled and linked with the Kitsune runtime system library to generate a shared object file. The new program source files plus the C code transformed *xf* files are the inputs for the compiler. The Kitsune building chain is composed by the Kitsune custom compiler *kitc*, the state transformer *xfgen* and the *GCC* GNU Compiler Collection.

Firstly, when the target application is executed, which shall be built with Kitsune, a DSU driver routine starts. This component launches the shared object file where the target program has been embedded. At the moment that a DSU request is received, the driver waits until the program reaches an update point. These points are manually specified by the developer and it is encouraged to place them on long-running loops of the programs, so

they can be reached periodically. The main execution thread jumps then to driver routine where the DSU process starts. First, the new program code is loaded and the entry point of it is invoked. State transformation functions are then executed, so the program state is consistent with the new updated code. The program is updated eagerly.

Kernel

A kernel is the core of an operating system, where the interaction between the user-space applications and the hardware is handled. It is the first code that is executed and it manages all the computer resources, such as memory or peripheral. Kernel-oriented DSU systems have commonly been oriented to general purpose operating systems. The analysed DSU system target UNIX-like ones.

LUCOS

LUCOS provides DSU capability for operating systems using virtualization [115], which was proposed by the same authors of POLUS [110]. Through virtualization, an extra layer is inserted between the hardware and the operating system, which is usually handled by a virtual machine monitor. This software element has the control of the execution and the state of the virtualized operating systems. The DSU process is carried out by the virtual machine monitor. For this, function redirection calls and state transferring tasks are performed. A prototype was built and tested, where the Linux kernel was dynamically updated from version 2.6.10 to 2.6.11. Four steps are commonly carried out: analysis of the static patch, dynamic patch source code generation, building an executable dynamic patch binary and dynamic patch injection and the DSU operation.

The Linux kernel permits to insert a new code on run-time through a loadable kernel module. This module is the LUCOS dynamic patch module, where changes to the executable instructions and data structures are specified. State transformation functions are also included, which convert the old data structure version to the new one. Furthermore, a register for callback, dynamic patch initialization and clean-up, and *module_init* and *module_exit* functions may also be enclosed. LUCOS provides a set of helper methods with the aim of guiding the developer while writing all these functions. At the time of writing, this DSU does not offer any automatic dynamic patch building toolchain or framework. Nevertheless, this work is considered as one of the tasks to be achieved for future work.

The DSU process in LUCOS is carried out in three main states. First of all, when a DSU request is received, the dynamic patch is validated and prepared. Before applying the dynamic patch, LUCOS verifies that those

threads, which are executing the code to be updated, are halted. In case that none of these threads are active, state transfer functions are invoked. The consistency between the old and new program state is managed by the virtual machine monitor. Whenever a write instruction is called from the program, both program states are synchronized. For the code transformation, a jump instruction is inserted at the beginning of the old executable program through binary rewriting. Finally, LUCOS inspects the stack to see if no old execution thread still resides on the stack. At this point, the DSU process is concluded.

DynAMOS

DynAMOS is a DSU system designed to perform dynamic kernel updates [116]. The core subsystems of the Linux kernel were dynamically updated, where the source code differences were not needed. It is based on the adaptive function cloning technique, where updates are not applied at the basic block level, but at the function level.

DynAMOS detects quiescent functions by inserting usage counters because none of the functions to be updated shall be idle on the stack. For this purpose, entries and exits for all the code to be updated are monitored and the stack is examined. For the code transformation, execution flow redirection mechanism through trampolines is employed. Nevertheless, the adaptive function cloning technique allows switching dynamically between multiple function versions. This switching decision is managed by the kernel itself, where, for example, the system workload could be measured to decide which version to execute. Because of this, the program state of the running program version needs to be synchronized and handled, so it does not lead to an inconsistent state. The new program code should not access the old incompatible program state. Locks such as semaphores are used to handle program state data accesses.

Shadow data structures are used for the state transformation. When new data types are encountered in the new program version, new variables are instantiated on different memory addresses. These memory locations are freed when the original ones are released. As far as update points are concerned, the developer is not required to manually define update points. This is decided by the system. Dynamic kernel patches are manually built by the developer using standard building utilities such as the *gcc* compiler. As future work, the authors plan to create an automatized dynamic kernel updates building system.

KSplice

KSplice allows applying patches to the operating system kernel on the fly, without the need of rebooting [117]. During the testing phase, 64 kernel patches were applied. As the result, 87.5 % (56 out of 64) were successfully applied without the mediation of the developer. The remaining patches (8 out of 64) required additional code to be written. These software updates correspond to the security patches for the Linux kernel from May 2005 to May 2008. On contrary to other DSU systems, Ksplice examines the old and the new program compiled object files and meta-data, instead of analysing the source code. This DSU system was acquired by Oracle in July 2011. Actually, a commercial dynamic Linux kernel updating service is offered by the Oracle Linux Premier Support [118]. The sucessors (presented publicly) of KSplice are *KGraft* and *KPatch*, developed by SUSE [1] and RedHat[2] respectively . *KGraft* is able to update or replace functions. For this, *ftrace* utility is used to jump upon old function versions. In contrast, *KPatch* sends a stop machine command to the kernel so, whenever it is safe, a halted condition is reached. At this point, the new code is injected. Nevertheless, changes to the internal data structures of the kernel are not possible neither in *KSplice*, *KGraft* nor *KPatch*. These two sucessors were unified when Red Hat created the *livepatch* system in 2014, which also uses the *ftrace* utilities as *kGraft*. *livepatch* is available at the mainstream kernel [118].

This kernel live patching functionality is self-contained, which means that it does not rely on or use any other kernel module. As indicated by *Chris Binnie*, this kernel functionality is only available for x86 architectures. Thus, this DSU is not still available for PowerPC or ARM [118]. KSplice works directly with binary files and the meta-data instead of program source code [117]. Thus, legacy binaries can be updated. Implementation limitations, safety issues and the involvement of the developer are also avoided or reduced. Nevertheless, this approach introduces new compilation related challenges, such as how to deal with optimized binaries which camouflage the purpose of the patch. Safe multi-threaded updates are not possible. This DSU mechanism replaces the entire function, in case that some part of it has changed. A new function version is linked by inserting a jump instruction at the beginning of the old version. The DSU process usually takes about 700 μs, in which the system is blocked. Firstly, *pre-post* differencing is used to build the object code for the upgrade. The *run-pre* matching is then employed in order to resolve symbols properly and to provide safety.

The modified code is identified by comparing the old and the new binary codes at the *pre-post* differencing. For this, the original kernel sources and

[1] `https://www.suse.com/`
[2] `https://www.redhat.com`

the patched ones are compiled, and the object files generated. Two compiler options are enabled (*-ffunction-sections* and *-fdata-sections*) to guarantee that the functions and the data structures are placed in their own section inside the binary. In case that these binary function instructions differ but although they provide the same behaviour, KSplice would replace them. This would be unnecessary itself, however, it does not compromise safety. After the comparison, the modified functions are extract and a DSU patch generated with this information. Even the replacement is ready, the symbols are required to be resolved. The information needed to perform this task is included in the DSU patch. At the *run-pre* matching phase A jump instruction is introduced at the start of the old program code, so the new code can be executed next time. This is performed in the kernel space for security reasons. As far as the update point is concerned, which is time instance when the DSU process starts, the *stop_ machine* utility is used by KSplice in order to end up to a safe update point. In case of multiple unsuccessful tries, the DSU process is discarded.

K42

K42 is a open source object-oriented operating system kernel which focuses on customizability, scalability and maintainability [119, 120, 121, 122]. Linux Application Programming Interface (API) and Application Binary Interface (ABI) are supported. Consequently, Linux libraries and applications can be executed in top of this kernel, as the replacement of the Linux ones. In a similar way to Linux read copy update, quiescent states are also detected. Nevertheless, an object translation table is used in K42, where each of the entry specifies the per-address space of each operating system object. Calls are performed through this table.

In this operating system, hot-swapping of objects is possible. For this purpose, incoming calls to the object are first temporally suspended. The program state is then transferred to the new object when a quiescent state is detected. The global reference is finally modified by changing the corresponding object translation table entry, and the incoming calls redirected, so the new objects are called and used at the next invocation. The DSU mechanism of K42 is based on this functionality.

The DSU feature was implemented and tested by *Andrew Baumann* [120, 122, 121]. In order to provide DSU capabilities, factory objects were added, which are responsible of creating and tracking dynamically updatable objects. When an object to be updated dynamically is desired, a factory object is first instantiated. If an old version of it already exits, it is updated itself before proceeding with the DSU process. However, the objects instances associated with the old factory need to be associated first to the new fac-

tory object. The factory object is then able to instantiate and hot-swap the desired objects. As far as state transfers are concerned, a state transfer protocol is used to transfer the state from the old objects to the new ones. A common intermediate format, which shall be understandable by both object versions, is employed. No automatic state transfer function generation is provided. Consequently, the developer is required to write all the needed code, which could be a laborious task in some cases. In contrast to other DSU systems, DSU process may be applied lazily, for example depending on the amount of instances of an object.

PROTEOS

PROTEOS is a research operating system, designed from the beginning to provide DSU capabilities [123, 124]. Through its DSU feature, upgrades at the process level are performed, instead of at the function level. An automatic state transfer is provided, which is used to transform and transport the old program data to the new one. State checking and hot roll-back functionalities are also given. PROTEOS, which is based on *Minix 3* [125] and is compatible with the Portable Operating System Interface (POSIX) interface, can be initiated on x86 architecture computers.

All the PROTEOS operating system kernel components are enclosed within processes. These kernel modules are for example the network stack, memory management, process management, scheduling and drivers. This architecture, shown in Figure 3.3, permits updating and replacing those modules independently, where the new code and program data is inserted. The kernel module which provides DSU capabilities is the *update manager* (UM), which is in fact able to update itself on the fly. Moreover, the microkernel provides interprocess communications, which are achieved by message passing. These messages are used for the communication and synchronization between the kernel processes and the UM. The hardware interface is also provided in this layer.

When a DSU request is received, the new program version is loaded as a new process instance in the memory. At this point, the update manager waits until all the remaining processes reach a safe update state. The old program process is then atomically replaced by the new one. The next step is to transfer the old program state to the new one. For this, a state transfer framework is provided, which first inspects the program state of the old and the new version and secondly performs type and memory transformations so the transferred state is compatible with the new program version. This process is based on the LLVM framework [126], through an instrumentation pass [126]. The new program process is ready to be executed and the old program process is deleted.

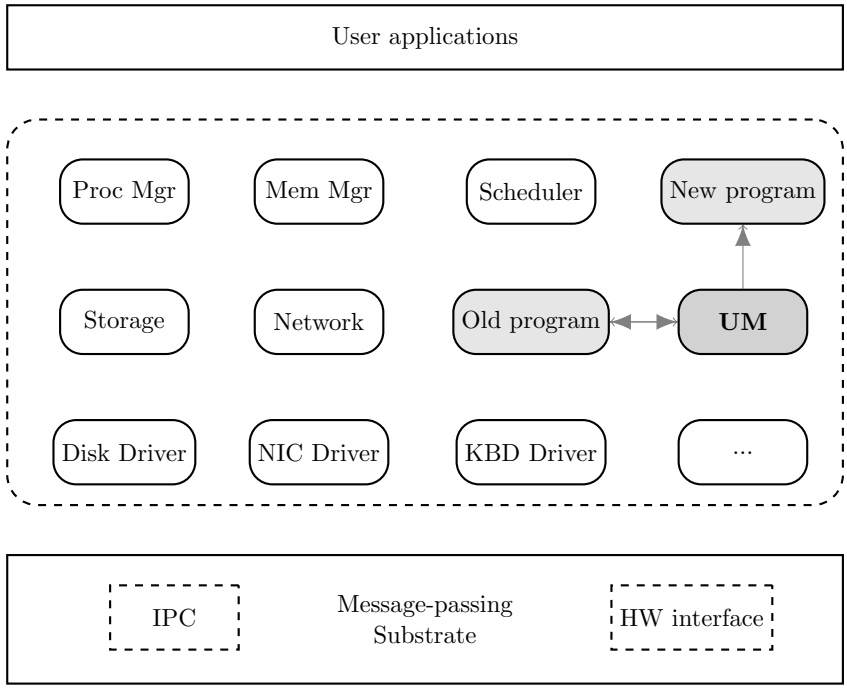


Figure 3.3: Architecture of PROTEOS [123, 124]

PROTEUS is able to monitor and analyse the state transfer process. In case of an anomaly or an error is detected, this DSU system is able to rollback, so it can recover and launch the old program. This is possible because the old process has not been modified during the state transfer process. A time-traveling state transfer mechanism was also presented [124, 127], where state transfers across three process instances are performed. These instances are: the old program version, the new program version and the reversed program version. The goal of the time-traveling state transfer is to provide a fault tolerant state transferring utility, suitable for a reliable DSU system. Apart from the usual forward state transfer between the old and the new program, a reverse conversion is performed from the new program one to the reversed version. As a result, deviations between the old and the reversed program states may indicate corruption or memory errors while the state transfer processes.

Real-time

In this group, in alignment with [106], DSU systems which were created for real-time, embedded system or IACSs are analysed.

DURTS

Dynamic Update for Real-Time Systems (DURTS) is able to produce and load the replacement code into the system and perform a dynamic update [128]. This DSU system was implemented on RT-Match, which is a UNIX-like RTOS. Two new operating system primitives were added in addition to the DSU features. Updates at the function level are performed, where the local state information is not conserved. Because of the timing constraints which the DSU system needs to meet, a schedulability analysis is performed, where the fault-tolerant rate monotonic scheduling algorithm is considered. The aim of this analysis is to ensure that timing deadlines are met while the DSU process. On average, a delay of 5.6 μ seconds was achieved on a Compaq Deskpro® 200 MHz Pentium Computer.

The DSU process consists of three main steps. First, a pseudo-linker is used to create the replacement code. Once the source file of the application to be updated is edited and compiled, the pseudo-linker creates a loadable module (using a standard linker, a binary file corresponding to the whole application would be generated). The pseudo-linker generates a pseudo-linked object file (**.o.plo*). This file contains the addresses for externally referenced symbols which are attached to the target application. Secondly, the new replacement code is loaded into the memory, in the heap section of the application memory, more specifically. Finally, the execution to the new replacement object is switched. For this, a pointer-to-function variable is used as a trampoline. Thus, at the next function invocation, the replacement code placed in the heap will be executed. In case of multi-threaded applications, additional steps are needed. Two new system calls were added (*rt_thread_set* and *rt_thread_get*) to the RT-Match operating system in order to change the process control structure of the thread. This information is modified to specify which should be the next execution instruction.

EmbedDSU

EmbedDSU, is a DSU framework for Java based smart cards, which is based on a modified SimpleRTJ embedded Java virtual machine [129]. An indirection table approach is used. However, multi-threaded applications are not supported in this embedded Java virtual machine. Thus, this kind of applications can not be deployed and neither dynamically updated.

The DSU is divided in two main steps. On the one hand, at the *off-card* phase, the differences between the classes of two versions are obtained through a DIFF generator. The result is a DIFF file, which is then parsed and transformed into a compressed binary format. This binary object is ready to be transferred to the smart card target. On the other hand, at the *on-card* phase, the binary DIFF file is loaded, where the digital signature

is checked and the information from it is extracted and interpreted. This information is passed to the *Patcher*, which is responsible of performing the dynamic update. The DSU process starts when a safe update point is detected. The modified Java virtual machine is then able to inspect and modify the required data structures, such as class meta-data, method bodies or object instances, to perform the desired update.

Gracioli

A operating system infrastructure for remote code updates was presented by *Gracioli* [130]. This infrastructure, which adds low-overhead, is based on the Embedded Parallel Operating System (EPOS). It is a multi-platform and component-based embedded operating system. In this operating system, an indirection level between components is possible. This characteristic is employed for the dynamic updates.

The updatable components are marked during the first compilation and building process. Once the operating system is initiated, the infrastructure is only able to update those components marked as updatable ones. When a DSU of a component is wanted, the invocation to the component passes through *Proxy*, where a indirection to the updated code is performed. A semaphore is used to ensure that the invoked component is not executed while it is being updated.

EcoDSU

EcoDSU (ETRI CPS open Dynamic Software Updater) is a DSU system designed for cyber-physical systems [131]. Whole program updates are performed. However, global and local data can be just partially updated. When a DSU process is desired to be performed, the application to be updated is halted at the update point. In this instance, the executable region of the program is replaced by the new version. By default, data and stack areas are not modified. A set of command line utilities are provided to control, monitor and customize the DSU process.

Seif-Real

An approach to perform DSU operations in real-time applications is presented [132]. The update of real-time tasks is considered, where the Rate Monotonic scheduling algorithm is used to assign and manage the different resources employed by those tasks, usually the CPU. A simulation experiment is performed to evaluate the approach. However, it has not been implemented yet, and neither tested through a case study.

A dummy task is created, which is executed every hyper-period and has the lower priority among tasks, does not perform any operation by default. Nevertheless, DSU operations are executed when a DSU request is received. In this case, it is assumed that the new task version may have a different execution time, period or deadline. Thus, a schedulability analysis is performed. For this, the set of tasks is gathered, the new hyper-period is calculated and the schedulabilty study is performed. If the schedulability criteria is not achieved, the DSU request is rejected. If met, the updated task is loaded into memory and the task execution points are resolved. As a result, the new updated task is executed at the next second hyper-period. As stated, in the worst case the DSU process is achieved at the starting of the next second hyper-period.

Wahler

An approach to apply DSU procedures on real-time systems is presented by *Wahler et al.* [133]. It is based on an underlying Commercial-Off-the-shelf (COTS) operating system, where features provided by it are used to support DSU capabilities. Three commercial operating systems were evaluated for this purpose: QNX Neutrino[3], VxWorks[4] and Integrity[5]. None of them provides DSU features directly. Nevertheless, a component-based framework infrastructure can be created in top of the operating system, where taking advantage of the features provided by the operating system, DSU features for those components may be supported. The framework is built upon components and channels, which provide communications among components through a message passing mechanism.

The system checks at the end of each execution cycle if any of the components needs to be updated. In case that the DSU process for these components takes more time than the available one, the DSU process is carried out among several execution cycles. The new component version is transferred to the target with low priority. This ensures that other component deadlines are fulfilled. In order to transform the actual running component into the new version, the message passing mechanism is used. The component manager changes the message passing channels configuration, so the information flow is redirected to the new component.

For the state transformation, a size-fixed shared memory is reserved. This area is where the terminating old component saves the actual program state, in addition to some meta-data. After that, this memory is accessed by the

[3] `http://www.qnx.com/content/qnx/en/products/neutrino-rtos/neutrino-rtos.html`

[4] `https://www.windriver.com/products/vxworks/`

[5] `http://www.ghs.com/products/rtos/integrity.html`

new component, so the information contained in it is used to compute the new program state. It is assumed that the component state is possible to be transferred within the same cycle while the DSU process is performed.

An improved solution was then presented by the same authors [134]. In this approach, the state of the new component is gradually constructed after the DSU component is loaded. An atomic operation is then performed when the whole state is transferred. This method permits performing dynamic updates to those components with large states, which could not be possible to achieve by the previous described method. This solution was implemented and tested on top of RT-Linux[6]. However, it could be applied on other POSIX-compatible operating system.

First of all, the new component is instantiated and initialized, and the corresponding channels for message passing are generated. The execution of this component is also scheduled. Secondly, the state transfer process begins. This DSU permits the transfer of large states, which may be accomplished in multiple execution cycles. However, due to the fact that the component state transfer may be altered or modified through those execution cycles, a state synchronization algorithm is needed. This algorithm tracks the modification occurred within the old component, so the correct state representation is transferred and visible for the new component.

Finally, once the state transfer and synchronization is performed, which means that both the old and the new components hold the same component state data, each of them in their respective valid representations, the last step of the DSU process starts. At this point, the system switches to the new configuration, where the new component will be used.

As concluded by *Wahler et al.* [134], the most important challenge resides on how to deal with the updated or updatable software involved on a certification process against a standard, for example the IEC 61508 [13]. A whole re-certification of the system would be needed in case any software update is performed.

FASA framework

FASA (Future Automation System Architecture) is a scalable component framework for distributed control systems, where the application components can be executed on a single core platform, on a multiple core platform or distributed among several platforms [135, 136, 137, 138]. DSU features are provided, where new versions of the components may be updated dynamically. Figure 3.4 shows the software stack of FASA deployed on a multi-core platform.

[6] `https://rt.wiki.kernel.org/index.php/Main_Page`

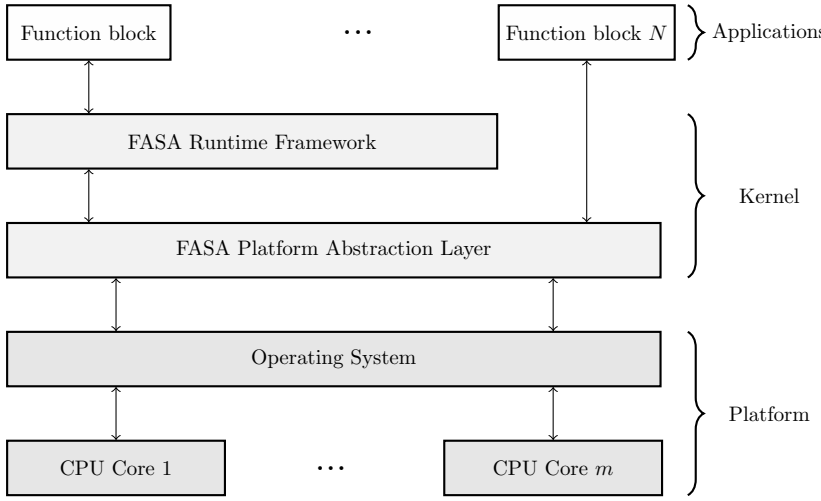


Figure 3.4: The FASA Software Stack on a multi-core platform

As it can be observed in Figure 3.4 above, the FASA framework is built on top of a commercial real-time operating system and provides an abstraction level to the application layer. This ensures that the execution of the application-oriented functions is transparent to the underlying operating system and platform [135, 136, 137]. After providing the software components to be executed by the developers, an initial deployment plan is calculated and the most suitable communication protocol is chosen. At this phase, the resource allocation, such as cores and hosts, is performed and the static schedule of the executions is determined [135].

The *Applications* layer of the FASA framework is composed upon the function blocks, where one or several of these function blocks may constitute a component. These function blocks are communicated each other through unidirectional channels, which means that a function block can either write or read on a single channel port. The channel implementation, which can be only used for data transmission, differs depending if the corresponding blocks are located on the same core, same platform or on a separated platform. All these blocks are executed cyclically at a fixed rate.

The DSU process is divided into five main phases [138, 139]. It has to be noted that, it is assumed that the functional and timing behaviour of the new component version has been already assessed before proceeding with the update. The process begins by loading the new component into memory as a shared library. Several ways to do it exist depending on the platform, such as through a serial communication, storage devices, debugging interfaces or network.

The next step is to prepare the system configuration, which describes the components, schedule and channels arranged in the system. A clone of the actual configuration is done, where a couple of modifications is then made. An instance of the new component plus the communication channels are created. In addition, a new schedule is created for the defined new component, so it can later on replace the old one. At this point, the state transfer phase starts. In case that no stateless information is used, this step can be safely omitted. On the contrary, state synchronization mechanism are employed to keep track of the state changes through the execution cycles. This is realized using two special function blocks in FASA, *Teach* and *Learn*. In addition, auxiliary monitoring components can be added to monitor and compare the behaviour of the new component against the old one. Finally, in order to complete DSU process, an atomic switchover is performed. This is accomplished by readjusting the system configuration. A common time stamp among platforms is used in order to ensure that all the states are up to date and the execution traces are synchronized. Rollback feature is also provided in FASA, which is able to restore the previous configuration in case of a safety violation.

3.2.1 Comparison

All the DSU systems targeting compiled applications perform dynamic updates in top of a UNIX-like operating system, usually GNU/Linux running on a x86 computer. Moreover, UpStare has been tested on Linux 2.4-2.6 running on a i386 architecture computer, Solaris 5.10 running on a SPARC computer and MAC OSX running on a PowerPC computer. However, none of these DSU systems provides real time features, which may be needed for an industrial control application. Except for DynSec, all the DSU systems targeting compiled applications present a dynamic patches building toolchain.

In case of DSU systems designed for operating system kernels, LUCOS, DynAMOS and kSplice (and its successors KGraph and KPatch) target the Linux kernel while K42 is the operating system kernel itself, designed to provide customizability, scalability and maintainability. PROTEOS is designed for the *MINIX 3* [125] operating system, which is compatible with the POSIX interface. In this DSU system, where process level upgrades are performed, automatic state transfers are provided. PROTEOS is executed on x86 computers and according to the author, the proposed technique can be easily applied to other operating systems, for example: L4, Integrity or QNX. To the best of our knowledge, kSplice and its successors, KPatch and KGraft mechanisms, are the unique DSU systems which are applied and offered commercially for the Linux kernel.

DURTS is the only real-time DSU targeting a UNIX-like operating system, specifically, the RT-Mach operating system. Other specific operating systems were used on EmbedDSU and Gracioli. While EcoDSU works purely on bare metal, without the support of an operating system, Wahler and FASA are operating system agnostics. This means that any operating system can be used. Nevertheless, they shall be POSIX-compatible. Wahler and FASA use operating system utilities to perform those software upgrades, where application components are updated.

Commonly, each of the DSU systems provides a dynamic patches building system, which is used to generate such dynamic patches. It has to be noted that, as stated by Hicks [75], the concept of a dynamic patch differs from the regular static patch from the software development implementation or maintenance phase (*diff* and *patch* commands on UNIX), where it is described as the differences between sources of two program versions. A dynamic patch can consist of new updated executable code, new global variables, type/state transformers and additional meta-data. This information is then transferred to the DSU runtime system, where after receiving a DSU request, the DSU process is carried out [88]. These systems most often provide a set of tools, such as custom-compilers, source-to-source patch generators or source analysers in order to create dynamic patches. CIL [113] and LLVM [126] compilation, source analysis and transformation tools are commonly utilized [82]. Table 3.7 shows the required tasks to be accomplished to generate a dynamic patch [75, 82]. The order in which these tasks are accomplished may vary among dynamic patches building systems.

Task	Description
1	Identification and syntactical comparison of the new program version against the old one(s).
2	Adaptation of the program (source or object) so it can be updatable.
3	Construct the type/state transformer functions (whenever possible).
4	DSU meta-data generation, where the information about current program version is defined.
5	Dynamic patch construction and/or assembly.

Table 3.7: Tasks for building dynamic patches

Many of the DSU systems provide utilities to create dynamic patches. The tool developed by *Hicks et al.* [75] analyses and calculates the differ-

ences between two source files. A dynamic patch is then generated with the information obtained from it, which reflects those source code modifications. By default, the tool tries to automatically construct the required state transformation functions. Nevertheless, the programmer might need to complete or adjust these methods if the task could not successfully be accomplished. Stub functions are also inserted, where call indirections are specified. In order to attest robustness and safety, a safe C-like language was used.

In OPUS [93], a static patch analysis is firstly performed to examine the safety attributes and the compatibility of such source code modifications. If the required evaluations are successfully met, the dynamic software patch binary is generated. Patch definition meta-data is also embedded in this file. Similarly, dynamic patches are created through a source-to-source compiler in POLUS [110] (referred as *patch constructor* by *Chen et al.*), which is capable of identifying the semantics and syntactic differences between two software versions. In POLUS, the whole-program dynamic patch is compressed on a single file, which is the most adopted approach.

After generating the dynamic patch source code through the patch generator, a custom compiler, based on the CIL framework [113], is used by *Makris* to create an executable update patch [112, 76]. It has to be noted that the programmer is usually required to adjust the state mapping functions and determine the update points. The patch generator and the custom compiler are part of the UpStare framework.

In the Ginseng DSU system, a patch generator and a compiler are used, which are both based on the CIL framework [113]. Although the patch generator is able to automatically generate type transformers, the programmer needs to specify and define the state transformations functions. Type wrappers are used, where new types with larger amount of memory are defined as the replacement for the old ones. Safety analyses are performed to ensure that safety properties are met while DSU tasks are carried out. In Kitsune, code transformations, state transformations and update point attribute need to be manually specified within the original program source [81]. For this purpose, the *xfgen* tool might be used. Besides, Kitsune uses standard compilation utilities.

Code transformation

Table 3.8 classifies the analysed DSU systems according to the **code transformation** properties. Employed *technique* and the upgraded *unit of update* are inspected. As shown in Table 3.8, most of the DSU systems employ trampolines (45 %) and indirection handling (30 %) techniques. Trampolines can be inserted by employing functions pointers or by rewriting the

register where the next invocation is stored. As a special case, in DynSec, software updates are performed indirectly by flushing the code cache. The patched instruction is first translated and then executed, while the original code is not altered. As it can be observed in Table 3.8, software components are commonly updated by these DSU mechanisms (35 %), while functions and whole-programs are upgraded 30 % and 25 %, respectively. In kernel-oriented systems, the operating system kernel is considered as the computer whole-program.

Name	**Code transformation**	
	Technique	**Unit of update**
DLpop	□	▼
OPUS	▽	▲
DynSec	△	▲
POLUS	▽	▲
UpStare	▽	■
Ginseng	□	■
Ekiden	*– not supported –*	
Kitsune	△	■
LUCOS	▽	■
DynAMOS	▽	▲
KSplice	▽	▲
K42	□	▼
PROTEOS	▽	▼
DURTS	▽	▲
EmbedDSU	□	▼
Gracioli	▽	▼
EcoDSU	△	■
Seif-Real	?	
Wahler	□	▼
FASA	□	▼

△ Binary rewriting ▲ Function
▽ Trampolines ▼ Component
□ Indirection handling ■ Whole program

Table 3.8: Code transformation properties of the analysed DSU systems

A distinction on the *unit of update* property definition is necessary. The *component* and *module* terms are widely used by the DSU system designers. However, the definition of them depends on how each of the author defines

them. The differentiation of these terms may be misunderstanding, since a module can be defined as a component and vice-versa. Consequently, in this work, a software component is referred to a piece of software, composed by one or more functions or methods. As it can be seen in Table 3.8, modules are commonly replaced by these DSU systems (45 %), while components and whole-programs 20 % and 25 % of the times, respectively.

State transformation

Table 3.9 shows the **state transformation** properties for each of the analysed dynamic software updating techniques listed in Table 3.6. *State transformer*, *mode* and *data update* properties are examined.

Name	State transformation		
	State transformer	**Mode**	**Data update**
DLpop	◁ ▷	▸	⊞
OPUS	◁	▸	⊞
DynSec	– *not supported* –		
POLUS	◁ ▷	◂	⊞
UpStare	◁ ▷	▸	∪
Ginseng	◁ ▷	◂	∪
Ekiden	▷	◂ ▸	∩
Kitsune	◁ ▷	▸	∩
LUCOS	▷	◂	⊞
DynAMOS	▷	◂	⊞
KSplice	– *not supported* –		
K42	▷	◂ ▸	∩
PROTEOS	◁	▸	∩
DURTS	– *not supported* –		
EmbedDSU	◁	▸	∪
Gracioli	– *not supported* –		
EcoDSU	◁	▸	∪
Seif-Real	?		
Wahler	▷	◂	∩
FASA	◁ ▷	◂ ▸	∩

◁ Automatic ◂ Lazily ∩ Checkpointing
▷ Manual ▸ Eagerly ∪ In-place
⊞ Indirection

Table 3.9: State transformation properties of the analysed DSU systems

As observed in Table 3.9, Dynsec, KSplice, DURTS and Gracioli do not provide **state transformation** features. Even that automatic state transformer generation and construction is possible, input from the developer is often required and/or advised. OPUS, PROTEOS, EmbedDSU and EcoDSU are the only DSU system, in which the state transformations are automatically handled. Lazy state transformations are possible in some DSU systems. In Ginseng, type transformers are not called at update time. These functions are invoked when the values to be updated are accessed. In Ekiden, it is up to the developer to decide and design if the state transfer should be performed lazily or not. On the contrary, in case of DSU systems targeting operating system kernels, LUCOS, DynAMOS and K42 allow the co-existence of old and new program data. A state synchronization procedure is then used to maintain the coherence between them. As stated by *Andrew Baumann* [121] DSU lazy transformations can also be performed in K42. A state synchronization algorithm is also used in FASA.

For the *data update* procedure, the described three methods are similarly adopted. While using the *in-place* approach, once the first version of the program data is located within the memory, it has to be verified that the new program representation data fits into the reserved memory. The *checkpointing* method provides the most versatile approach. Nevertheless, it is the most difficult one when it comes to the design and development of applications, since procedures to serialize and de-serialize need to be prepared and integrated within. Ekiden provides a state transfer updating library which enables serialization and de-serialization tasks.

Update point

Finally, as far as the **update point** is concerned, most of the DSU systems use the *Activeness Safety* approach to determine the update instance, which means that the old program which will be updated shall not be active. Two common approaches are used when an underlying operating system or virtualization layer exists: invoke a *stop_ machine* system call or stop the execution of the virtual machine, as done by LUCOS. For compiled applications, *manual* specification is also adopted. In this case, it is the responsibility of the developer to specify safe update points. Usually, this instruction is inserted in long-running loops, where no resources are held. Only Ginseng and DynAMOS follow the *Con-Freeness Safety* approach.

All the DSU systems targeting compiled applications except DLpop and Ekiden allow the upgrade of multi-threaded applications. In DSU systems targeting operating system kernels, neither KSplice or K42 provides multi-threaded kernel module updates. With respect to PROTEOS, a single kernel module process, which is the unit of update, may contain apart from the

main control thread, additional auxiliary execution threads. When it comes to the DSU systems targeting real-time control systems, in DURTS the underlying RTOS was modified to support this feature. Apart from DURTS, none of the examined real-time embedded system DSU systems provides multi-threaded program upgrade capacities. Notice that in Wahler and FASA several processes can be updated simultaneously in parallel taking advantage of the underlying commercial RTOS.

Table 3.10 depicts the **update point** properties of the analysed DSU systems:

Name	**Update point**	
	Specification	**Multi-threaded**
DLpop	⊠	No
OPUS	⋒	Yes
DynSec	⊠	
POLUS	⊠	
UpStare	⊠	
Ginseng	⊠	
Ekiden	⊠	No
Kitsune	⊠	Yes
LUCOS	⋓	Yes
DynAMOS	⊠	No
KSplice	⋒	
K42	⋒	Yes
PROTEOS	⋒	
DURTS	⋒	Yes
EmbedDSU	⋒	No
Gracioli	⋒	
EcoDSU	⊠	
Seif-Real	⋒	
Wahler	⋒	
FASA	⋒	

⋒ Activeness Safety (AS)
⋓ Con-Freeness Safety (CFS)
⊠ Manual

Table 3.10: Update point properties of the analysed DSU systems

Table 3.11 provides a summary of the analysed DSU system categorization and classification.

Name	Target	Date	Code transformation		State transformation			Update point	
			Technique	Unit of update	State transfer	Mode	Data update	Specification	Multi-threaded
DLpop	Application	2001	□	▼	◁ ▷	▸	⊞	⊠	No
OPUS		2005	▽	▲	◁	▸	⊞	⋒	Yes
DynSec		2013	△	▲	– *not supported* –			⊠	
POLUS		2007	▽	▲	◁ ▷	◂	⊞	⊠	
UpStare		2009	▽	■	◁ ▷	▸	∪	⊠	
Ginseng		2008	□	■	◁ ▷	◂	∪	⊠	
Ekiden		2011	– *not supported* –		▷	◂ ▸	∩	⊠	No
Kitsune		2012	△	■	◁ ▷	▸	∩	⊠	Yes
LUCOS	Kernel	2006	▽	■	▷	◂	⊞	⋓	Yes
DynAMOS		2007	▽	▲	▷	◂	⊞	⊠	No
KSplice		2009	▽	▲	– *not supported* –			⋒	
K42		2006	□	▼	▷	◂ ▸	∩	⋒	Yes
PROTEOS		2013	▽	▼	◁	▸	∩	⋒	
DURTS	Real-time	2004	▽	▲	– *not supported* –			⋒	Yes
EmbedDSU		2011	□	▼	◁	▸	∪	⋒	No
Gracioli		2014	▽	▼	– *not supported* –			⋒	
EcoDSU		2008	△	■	◁	▸	∪	⊠	
Seif-Real		2009	?		?			⋒	
Wahler		2009	□	▼	▷	◂	∩	⋒	
FASA		2014	□	▼	◁ ▷	◂ ▸	∩	⋒	

Technique:
△ Binary rewriting
▽ Trampolines
□ Indirection handling

Unit of update:
▲ Component
▼ Module
■ Whole program

State transformer:
◁ Automatic
▷ Manual

Mode:
◂ Lazily
▸ Eagerly

Data update:
∩ Checkpointing
∪ In-place
⊞ Indirection

Specification:
⋒ AS
⋓ CFS
⊠ Manual

Table 3.11: Summary and classification of existing DSU systems

3.2.2 Requirements for Safety & Security Compliance

None of the analysed DSU system provide an industrial safety-compliant solution. As stated by *Wahler et al.* [134], the biggest challenge for updating the software of real-time systems is that they often need to be certified according to standards such as the IEC 61508 [13]. Commonly, if the software of such system is updated, the system as a whole must be re-certified before it can be used in the field. Most of the DSU systems target or employ a UNIX-like operating system, which are not advised for safety-related systems, even that initiatives to evaluate and assess the Linux kernel against safety standards exist [140, 141].

Wahler, FASA and PROTEOS systems are the closest proposed solutions towards a safety-compliant DSU technique, since as stated by the authors, the described techniques are compatible with any POSIX-compliant operating system, where a safety certified one could be chosen. In case of PROTEOS, the core of the operating system needs to be modified. Consequently, after the changes, it shall be re-certified. Nevertheless, Wahler and FASA take advantage of the operating system utilities, so the DSU is transparent to the underlying operating system. The application, libraries and the dynamic updating mechanism running on top of the operating system shall be then certified.

Most of the studied systems do not provide protection against patching failures and the spread of such faults through the system, which might disturb and/or disrupt safety-related software components (or other non-safety-related ones), as well as compromising the availability of the system. Protection mechanisms are necessary to protect from patching failures and/or systematic failures of the new software version, which may for example monopolize the CPU or access unauthorised memory regions. At the application level, only DynSec provides a sandbox execution environment, for which a virtualization layer is employed. This technique was also used in LUCOS for the dynamic update of the Linux kernel. In PROTEOS, the new program version is loaded as a new process instance in the memory. This DSU system is able to perform a rollback in case of an anomaly or an error is detected during the patching process. Temporal independence could be achieved through the enforcement of an adequate scheduling policy. The solution proposed in Wahler might provide the protection against systematic patching failures, but it depends on the selected underlying RTOS and taken and/or implemented additional measures.

Fault monitoring and detection measures are widely adopted and implemented in safety-critical systems [13]. Software updates monitoring features were only employed in DynSec, PROTEOS and FASA. On the one hand, system calls are monitored in DynSec. On the other hand, PROTEOS is able to

monitor and analyse the state transfer process. Lastly, auxiliary monitoring components can be added to monitor, compare and analyse the properties and behaviour of the new component against the old one in FASA. However, remote monitoring features are not included by default. In the presented DSU systems, except in FASA, monitoring activities are performed within the same computer system and are automatized. Although, the separation between the monitor and monitored computer is highly recommended by the safety IEC 61508 [13] standard.

In addition, dynamic reconfiguration techniques are not recommended for the development of safety-critical systems [13]. A static resource allocation approach is advised. In these lines, the required hardware resources, specially execution time and memory, should be bounded before, during and after the DSU process. In Wahler, enough free memory must be ensured in the system to load the new component. Moreover, the CPU time employed by the main application should be low enough to enable the DSU service while executing it. Execution time and memory should be then initially assessed. In FASA, the system switches to a new configuration in the final phase. The hardware resources to be used by the new software component are not initially statically assigned. This characteristic introduces challenges for the compliance of the system against the safety standards, such as the IEC 61508 [13].

One of the requirements for the safety assessment (for any cyber-physical system in general) is the schedulability analysis, where it is ensured that tasks are executed and completed according to their timing requirements. The temporal delay caused by the DSU and its possible effects are not usually considered. Temporal constraints and requirements due to DSU mechanisms were analysed by Seif-Real [132]. As stated, the upgrade procedure should not cause any deadline to be missed. Therefore, the dynamic software updating service (enabled by a dummy task), performs first a schedulability analysis of the system upon the request of a software upgrade. If the new tasks are not schedulable, the update process is aborted. Just the main idea was presented in Seif-Real, without specifying the **code transformation**, **state transformation** and **update point** properties. These temporal issues were also taken into account by DURTS. Besides, in Wahler, in order to ensure that other software component deadlines are met, the new component is transferred to the target with low priority. However, during the real-time update of the component, the component manager shall replace the old one with the new one within a single execution cycle. If this condition is not met, the deadlines of other application components might be missed. In FASA, it is assumed that the functional and timing behaviour of the new component version has already been evaluated before proceeding with the update.

Regarding security, in order to fulfil with the requirements and guidelines specified by the IEC 62443 standard [14, 15], even adopting a DSU system, a fully replicated system would be needed to firstly verify the robustness and reliability of the DSU mechanism, to make sure that this process does not compromise the safety and temporal properties of the system, and secondly, to ensure the correctness of the new software patch. According to the standard [15], software patches should be tested and authorised by the asset owner before effecting and being installed in the production system. Nevertheless, it might not be possible to replicate the whole production environment or compose an accurate testing benchmark to test and validate a given software update. As far as the security of the dynamic patch is concerned, none of the systems introduces measures to ensure the integrity and confidentiality of it. Access control measures for system users are neither taken. An attacker could replace the security-related component with a dummy one, and dodge then already implemented security countermeasures.

Table 3.12 shows the main requirements for enabling DSU in safe and secure systems.

ID	Definition	Source
R_1	Protection against patching failures	IEC 61508 [13]
R_2	Software updates fault monitoring	IEC 61508 [13]
R_3	Static resource allocation	IEC 61508 [13]
R_4	Time-triggered architecture	IEC 61508 [13]
R_5	Internal testing capability	IEC 62443 [14, 15]
R_6	Software patch integrity and confidentiality	IEC 62443 [14]
R_7	Authentication and authorization of system users	IEC 62443 [14]

Table 3.12: DSU requirements for safety and security compliance

4 Cetratus

Cetratus is a DSUs enabled runtime framework for safety-critical systems, which is founded upon a quarantine-mode execution and monitoring mode. This approach, similar to the sandboxing technique widely used in the security field [142], aims at protecting the computer program, and therefore the system, against new possible security bugs or weaknesses introduced in the new software patch (R_1 in Table 3.12). In line with the ENISA [19, 20], software and patching malfunctions and breakdowns are taken into account. Patching failures due to patch incompatibility issues are also avoided. Figure 4.1 illustrates the DSU-featured actors of the system respect to dynamic software updates.

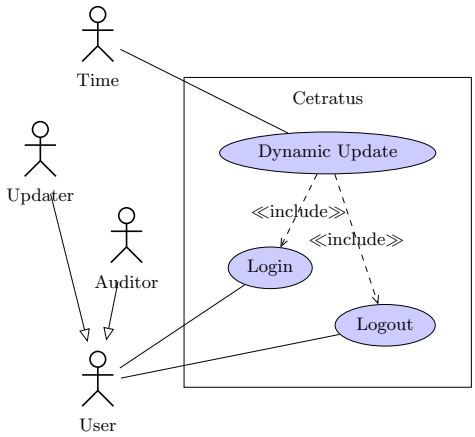


Figure 4.1: DSU-featured actors

As represented in Figure 4.1, two different system users have been defined: the *Auditor* and the *Updater*. Both the *Auditor* and the *Updater* inherit the *User*, which is a generic actor. Any system user shall be authenticated and logged in, in order to use the live patching features of the system, if allowed (R_7 in Table 3.12). The *Updater* is the system developer and/or maintainer. The dynamic update process is initiated by this user. On the contrary, the *Auditor* refers to the system examiner who shall check and validate the trustworthiness of the updated system. This is accomplished through the quarantine-mode execution and monitoring. The *Auditor* is able to revoke the dynamic software upgrade if the required system behaviour and/or per-

formance is not achieved. All the DSU process is also accomplished taking into account the real-time demands of the system (R_4 in Table 3.12). Figure 4.2 illustrates the dynamic update use case sequence diagram.

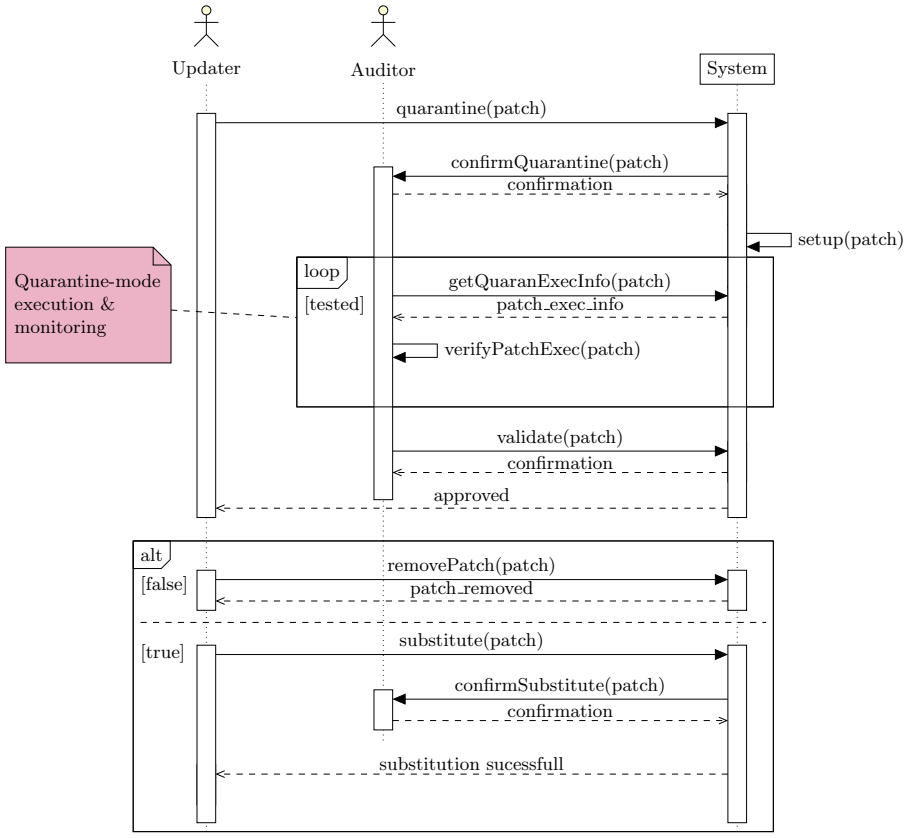


Figure 4.2: Dynamic update sequence diagram

The system first initializes the software patch, where code and state transformations are performed. At this point, the new software is ready to be run within the quarantine-mode, similar to a sandbox isolation. The new software version is then executed and monitored at the quarantine-mode phase (R_2 in Table 3.12). In case enough trustworthiness of the new software version is determined by the *Auditor*, the substitution is performed, where a role change between the old and the new software is performed. For this purpose, a version consistency strategy, as proposed by *Baresi et al.* [143], could be adopted. A ratification from the *Auditor* is then necessary in order to successfully proceed and succeed a dynamic software updating process. This user shall validate that new security functionalities behave as expected while the safety integrity level is not compromised [13].

Cetratus, in which application component upgrades are performed, uses a similar approach adopted by *P. Hosek* and *C. Cadar* [80] and *Wahler et al.* [139]. A multi-version execution strategy is adopted. Nevertheless, in *Cetratus*, while quarantine-mode execution, the output of the old program is always selected. In contrast to *Wahler et al.* [139], and in alignment with the ENISA [19, 20], possible software and patching failures are considered in *Cetratus*. The quarantine-mode aims at protecting the system against those events. Possible software patching failures are contained within the quarantine-mode (R_1 in Table 3.12). This technique prevents any propagation of faults through the system. New software faults while running, in addition to possible software patch setup errors at the *setup*(*patch*) step, are also kept under control. These failures shall be then reported by the diagnostics component.

Cetratus fits in with current industrial safety and security standards with respect to software updates [15]. The *Cetratus* dynamic update use case, illustrated in Figure 4.2, fulfils at the methodological and procedural level the IACS patching procedure proposed for the asset owner by the IEC 62443-2-3 technical document [15]. Indeed, the quarantine-mode execution and monitoring approach introduced in *Cetratus* accomplishes the *In internal test* activity required by the IEC 62443-2-3 [15], without compromising and/or affecting the whole system safety and/or timing properties or requiring a replicated system to reproduce the cyber-physical environment. Through *Cetratus*, this step is internally performed inside the system (R_5 in Table 3.12). As recommended by the standard [15], in order to enforce trustworthiness, a third-party authorization which endorses the software upgrade is required to perform such updates.

4.1 Design

The *Cetratus* system architecture for an application, which is composed by three upgradable components, is shown in Figure 4.3, in which the amount of application components are specified at the system design stage. The software architecture is divided in two parts. At the top, application specific components are provided. At the bottom, the *Cetratus* framework components are defined (depicted in yellow color in Figure 4.3). These elements are generic and reusable for any kind of application. For each of the upgradable application components, two containers are configured, A and B respectively (depicted in blue and green color in Figure 4.3). These containers are defined while system design and development phase, and might statically be allocated (R_3 in Table 3.12). A modular approach is also adopted, as recommended by the safety standards [13].

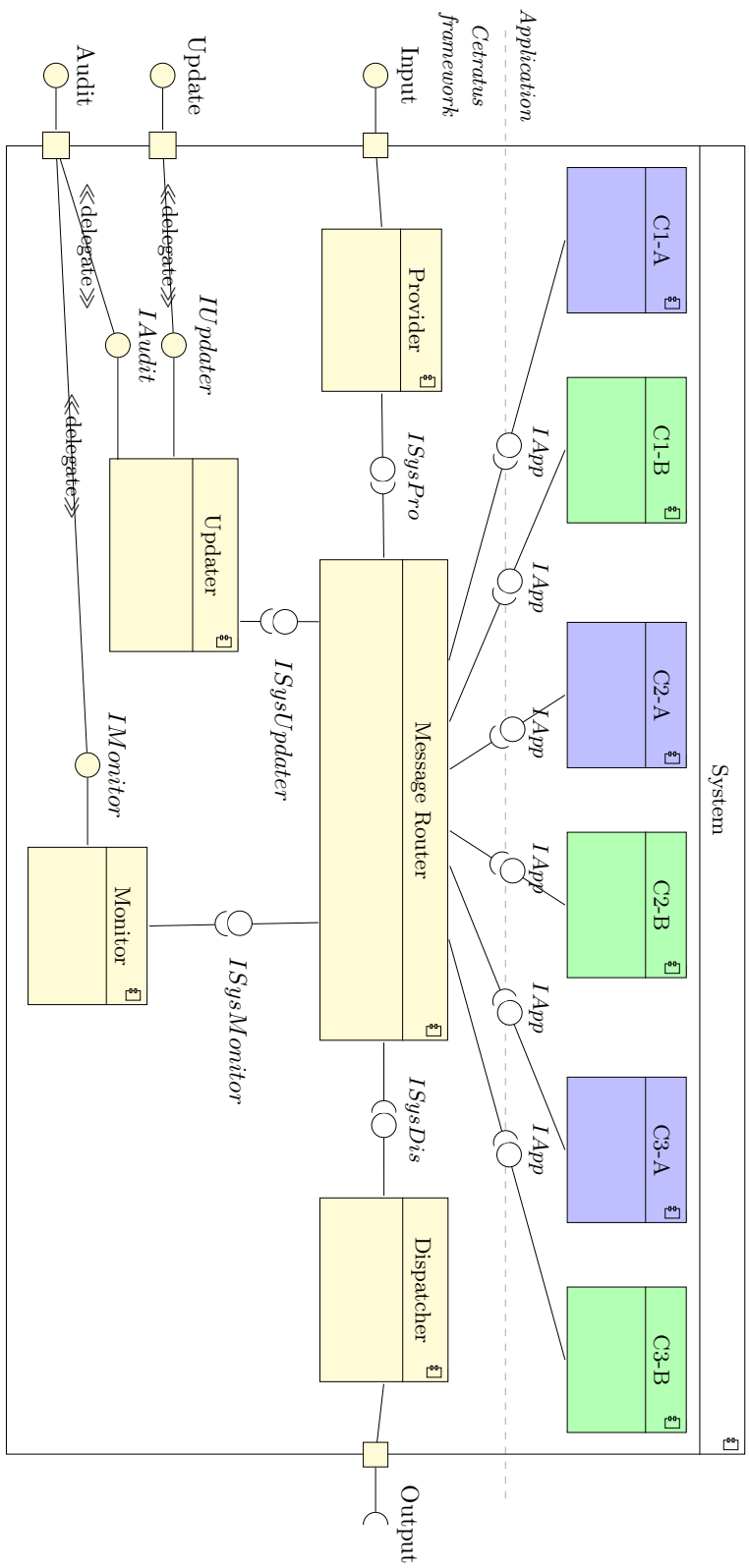


Figure 4.3: *Cetratus* runtime & system architecture for three upgradable component based application

Partitioning techniques are adopted in *Cetratus* for the execution of application components and for the quarantine-mode execution and monitoring of software patches. Each of the container is defined as a partition. For example, as depicted in Figure 4.3, six partitioned containers are created for three application components. These partitioning measures, widely used in mixed-criticality systems [53], provide independence of execution both in the spatial and time domains. Fault containment is then achieved (R_1 in Table 3.12). As far as live updates are concerned, the propagation of patching failures across the system are prevented. During the quarantine-mode execution and monitoring phase, new software faults while running are also kept under control.

A message-based communication is used among system runtime modules and application components, which are handled by the *Message Router*. System input data is provided to the application components by the *Provider* module and system outputs dispatched out by the *Dispatcher*. Apart from that, the quarantine-mode based dynamic software updating procedure is handled by the *Updater* module, while the *Monitor* provides the software patch execution data to the *Auditor* user (R_2 in Table 3.12).

4.1.1 Cetratus Framework Components

In this subsection the *Cetratus* framework components depicted in Figure 4.3 are described. The implemented interfaces by these modules are also provided.

Provider & Dispatcher

The *Provider* and *Dispatcher* runtime modules provide system input and output management services to the application components. They act as wrappers to the underlying specific input and output drivers, such as digital or analogous I/O, fieldbus communications, etc. System input information is fetched by the *Provider* from the data sources. In contrast, the *Dispatcher* manages system output communications and actuators. In a similar way to safety or security payloads produced by the application modules and forwarded to the *Dispatcher* are analogously handled.

In case safety-related or security-related digital communications are employed, the transportation layer of such data is just rearranged. Safety or security associated data payload is firstly acquired from the physical datagrams (for example from a PROFIsafe, Ethernet POWERLINK safety or Safety-over-EtherCAT fieldbus) and embedded then within an inter-partition message. This message will be next forwarded to the corresponding application component in which such communications are handled and processed. A black channel approach, widely employed for safety-related

communications, is adopted, in which security measures could also be implemented [17]. Figure 4.4 shows the *ISysPro* and *ISysDis* interfaces provided respectively by the *Provider* and *Dispatcher* modules.

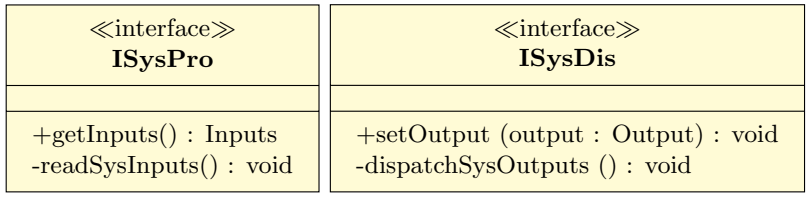


Figure 4.4: *ISysPro* and *ISysDis* interfaces

Message Router

The *Message Router* enables the information exchange among all application components and other system modules. All inter-partition communications are handled by this element. An ARINC-653 [144] inter-partition communication model has been adopted, where a message-passing mechanism is used. Firstly, communication services among application components and I/O wrappers (*Provider* and *Dispatcher*) are offered. Secondly, message duplications and redirections tasks are accomplished for the quarantine-mode execution and monitoring.

Two different interfaces are implemented by the *Message Router*. On the one hand, the *ISysUpdater* interface, shown in Figure 4.5 is presented to the *Updater* module. This interface offers to the Updater the necessary means for the interaction of it with other system modules. The *ComID* is an identifier label used within the system which specifies in which application component is the operation performed. It is extracted from the dynamic patch meta-data information.

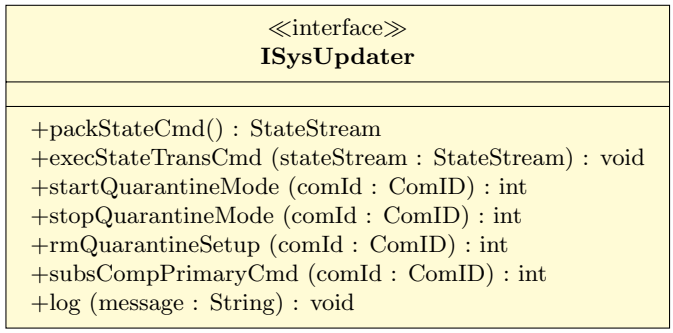


Figure 4.5: *ISysUpdater* interface

On the other hand, the $ISysMonitor$ interface shown in Figure 4.6 enables the quarantine-mode monitoring. Quarantine-mode patch execution information is retrieved through this interface while running by the *Monitor* module. The information supplied from the application component and the outcome produced by it are collected. This information is then supplied to the *Auditor*. The $QuaranInputInfo$ and $QuaranOutputInfo$ data objects contain respectively, input and output information transmitted and received from the application component while the quarantine-mode execution is accomplished. This data will be then supplied to the *Auditor*.

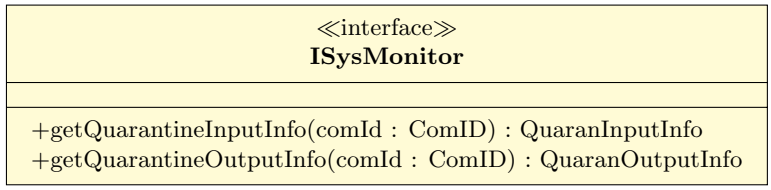


Figure 4.6: $ISysMonitor$ interface

Updater & Monitor

The quarantine-mode based dynamic software updating procedure is handled itself by the *Updater* module, while the *Monitor* provides software patch execution data to the *Auditor* system user. The *Updater* and *Monitor* modules, as depicted in Figure 4.3, enable the interaction of the Updater and *Auditor* users with the system. Nevertheless, an access control scheme regulates which user is allowed to perform given operations in the system.

The dynamic software updating process is managed by the *Updater* module, where an indirection handling table is used. This factory design pattern was also employed by *Baummann* in the K42 operating system kernel [121]. In this table, records associated to each of the application components are registered. Application component meta-data (name of the component, version and description) is stored and the status of the application components supervised.

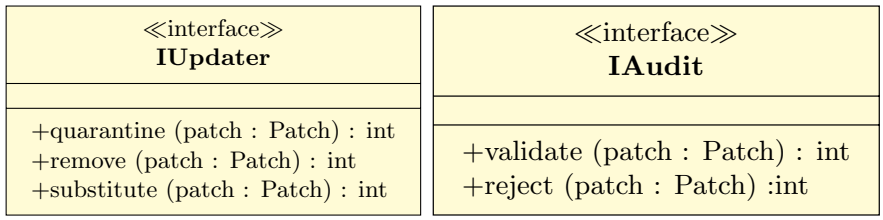


Figure 4.7: $IUpdater$ and $IAudit$ interfaces

In contrast, the Monitor collects quarantine-mode execution monitoring information for the *Auditor*, in which the input supplied to application component and the computed output from it is gathered. This is accomplished through the $IMonitor$ interface, which is shown in Figure 4.8. In addition, the execution footprint (such as CPU usage, timings and the required memory) evidences could be also obtained. This information might also be included in the $Patch\_exec\_info$ data structure. The diverse monitoring approach with separation between the monitor and the monitored system is highly recommended by the IEC 61508 [13].

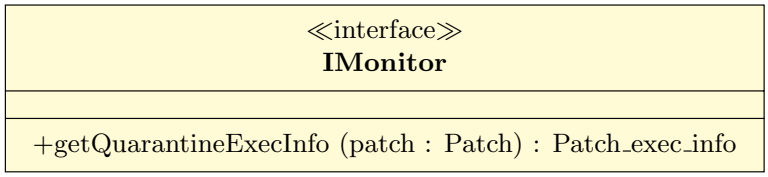


Figure 4.8: $IMonitor$ interface

4.1.2 Application Components

Application components are the basics of the safety-critical program, which may execute or offer many types of functions or services, such as security, safety, diagnostics, communications stacks or HMIs. Once the system is deployed, a replacement component can be created and a dynamic patch produced. In a similar way to hardware redundancy, these A and B component containers are alternatively switched as the primary application component racks. The second container is adopted as the secondary one, more specifically, on the quarantine-mode execution and monitoring stage. Despite, a secondary container would be not necessary if an application component is defined as not upgradable. The system shall hold enough resources for the allocation and execution of future software updates. Thus, most likely, it might initially be oversized.

Every application component shall fulfil with a predefined $IApp$ API contract. This interface compliance is mandatory in order to incorporate and couple new software components within the system. The application component shall be self-contained, single-threaded and not hold any hardware resource. This interface is depicted in Figure 4.9. Moreover, state packing and unpacking related operations depicted in Figure 4.9 would be not necessary in case stateless software design patterns are used for the design and development of application components. This strategy, which is encouraged by safety guidelines [13], reduces the number of steps to be accomplished in the DSU process.

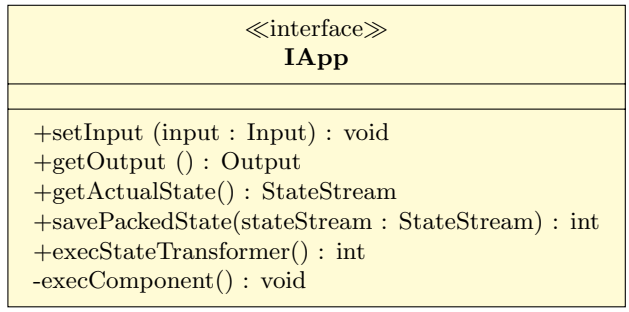


Figure 4.9: *IApp* interface diagram

In order to perform the state transformation, the actual state is packed and a *StateStream*, which is a binary encoded string, composed through the *getActualState*() method. In contrast to the *Input* and *Output* data objects, the state contains application component specific information, which might not be public for other components, for example the state of a communication protocol, Proportional Integral Derivative (PID) parameter values or application or component configuration. The application components can also employ the inter-component communication services offered by the *Message Router*. The *IComm* interface is offered to the application components for inter-component, as well as and other system module communications. This interface defines the methods to create, open and close a communication channel from which messages are sent and received. This interface is not depicted in the software architecture shown in Figure 4.3.

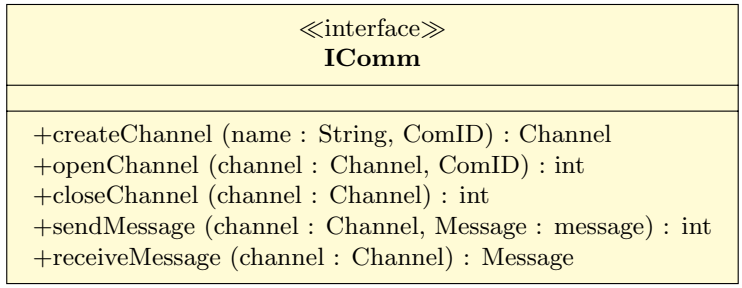


Figure 4.10: *IComm* interface

On the contrary to a static patch (*diff* and *patch* utilities on UNIX), as pointed out by *Hicks* [75], a dynamic patch may include, in addition to the new updated executable binary, state transformation procedures (new global variables, type and data transformers) and meta-data. A patch description file is described and used by *Hicks* [75]. This file contains the description and the implementation of the patch.

In *Cetratus*, the name of the application component to be updated and its version is supplied in the dynamic patch meta-data information. Code and state transformation procedures are also included in this file. Typically, a dynamic patches building system is needed, where source code compilation, analysis and transformation utilities are often utilized [82]. Static analysis are also frequently performed. Figure 4.11 shows the *Cetratus* dynamic patch building procedure.

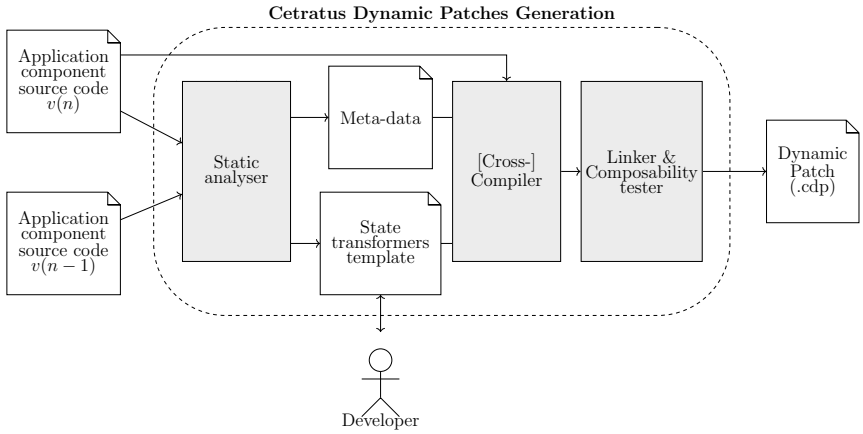


Figure 4.11: *Cetratus* dynamic patch building procedure

As depicted, a static analysis is initially performed. Through the semantics and syntactic analysis, the application component meta-data is extracted and the state transformation template created by looking and examining accessible variables. To this end, annotations are inserted by the *Developer* in the source code. The *Updater* user shall verify the generated state transformation functions, most likely, adjust and/or complete them. Additional static analyses might be also carried out that attempt to highlight possible security-related bugs and/or vulnerabilities. The dynamic patch is then compiled and linked. After linking, the composability of the new application component is checked. More specifically, the compliance of the generated dynamic patch against the *IApp* and *IComm* APIs is validated.

The information contained within the dynamic patch shall be protected when transmitted to the target system (through reverse-engineering, an attacker could obtain valuable information regarding system weaknesses and/or embedded malicious code). For this purpose, a secure communication protocol, such as the TLS, is necessary. In addition, a digital signature is created. The dynamic patch must be then transmitted together with this digital signature (R_6 in Table 3.12). These measures would ensure its integrity and confidentiality.

4.2 Safe Live Updates

In this subsection, the quarantine-mode based live patching approach employed in *Cetratus* is described. This sandboxing strategy was also applied by *Payer et al.* in DynSec [109] and suggested as future work by *Chen* [115] for LUCOS. In a similar way to the approach adopted by *P. Hosek* and *C. Cadar* [80] and *Wahler et al.* [139], a multi-version execution tactics is employed in *Cetratus*. In contrast to *Wahler et al.* [139] and FASA [139], the inclusion of new application components in *Cetratus* is only possible if unoccupied containers are available in the system. In FASA, where a whole system reconfiguration is feasible, system flexibility and reconfigurability is prioritized over safety. A roll-back mechanism is also provided in case unexpected behaviour is encountered in the new software version. However, neither dynamic reconfigurations nor backward recoveries are advised in safety-critical systems [13]. In *Cetratus*, such features are not present.

Figure 4.12 shows the quarantine-mode based dynamic software updating process time-line introduced in *Cetratus*.

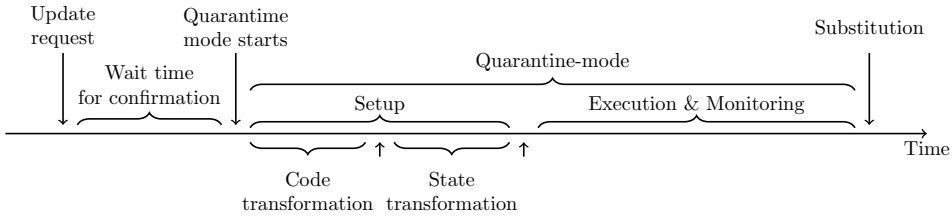


Figure 4.12: Quarantine-mode based dynamic software updating process time-line

Concerning real-time properties, the dynamic software updating process shall not alter or disrupt the services or performance of the system. Usually, the updating procedure stops the actual running program in order to apply the corresponding code and state transformation procedures. In case demanding transformations are faced or a constrained system is employed, this interruption might lead to a disruption of the service. Accordingly, in *Cetratus*, spare time (previously assigned) is employed to complete those transformations at the *Setup* stage.

As described previously, two containers are initially configured for each of the application components: A and B. These containers, which are both spatially and temporally isolated, are alternatively switched for the current executing application component and the possible quarantine-mode execution. Current primary container, which indicates in which of the containers the stable application component version is placed, is recorded in the *PRIMARY* state variable.

Figure 4.13 illustrates the *PRIMARY* state-transition diagram. As depicted, the A container is selected as the primary one when the system is initially deployed. The *PRIMARY* state variable specifies which of the A and B containers is actually employed as the primary one.

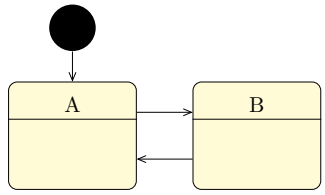


Figure 4.13: Application component patching *PRIMARY* state-transition diagram

Furthermore, an application component goes from a series of states during the dynamic software update process. This information is stored in the *STATUS* state variable and is aligned with the patch lifecycle model provided by the IEC 6244-3-2 technical document [15]. Figure 4.14 depicts the system *STATUS* state-transition model.

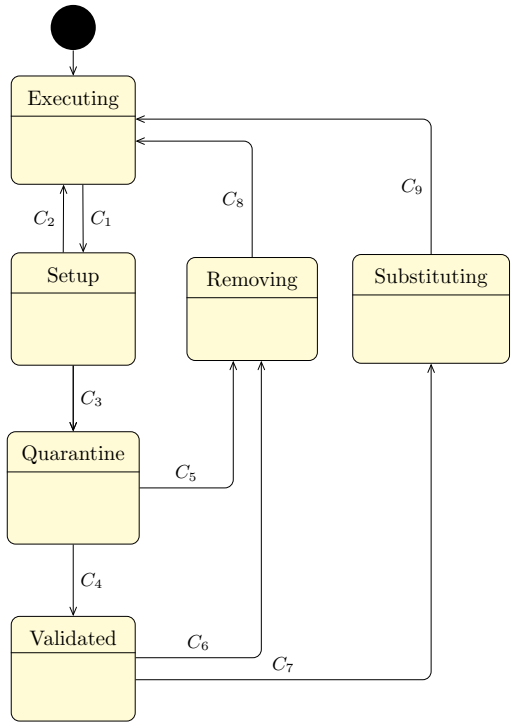


Figure 4.14: Application component patching *STATUS* state-transition diagram

Table 4.1 provides the description for each of the conditions shown in Figure 4.14.

ID	Description
C_1	A dynamic software update has been requested by the *Updater* and endorsed by the *Auditor*.The $quarantine(patch : Patch)$ method, available in the *IUpdater* interface shown in Figure 4.7 is used.
C_2	A time out exception occurs while trying to set up the dynamic patch. The dynamic software updating process is then aborted.
C_3	The setup has successfully been accomplished and the quarantine-mode based execution and monitoring starts.
C_4	The dynamic patch is validated by the *Auditor*. The $validate(patch : Patch)$ method from the $IAudit$ interface is used.
C_5	The dynamic patch is removed by the *Updater* or rejected by the *Auditor*.This is accomplished through the $remove(patch : Patch)$ (in interface *IUpdater*) or $reject(patch : Patch)$ (in interface IAudit) methods shown in Figure 4.7.
C_6	The dynamic patch is rejected by the *Auditor*. The $reject(patch : Patch)$ method is used.
C_7	A substitution request is received from the *Updater* and confirmed by the *Auditor*. The $substitute(patch : Patch)$ method (in *IUpdater* interface) is invoked.
C_8	The dynamic patch has been successfully removed.
C_9	The dynamic patch has been substituted. The $PRIMARY$ state variable is switched.

Table 4.1: Transition conditions for the application component *STATUS* states

Application component *PRIMARY* and *STATUS* states information, depicted in Figure 4.13 and Figure 4.14, are used and handled by the *Updater* module for the patching process.

The quarantine-mode execution and monitoring enables an isolated software patch execution, where software patching failures are contained. This method stops any propagation of failure through the system, likewise a mixed-criticality [53] approach. New software and patching defects are not considered by *Wahler et al.* [133] and in FASA [139]. These failures are kept under control through the quarantine-mode. Moreover, the live patching process is aborted and the patch rejected in case insufficient resources (memory and time) are faced. The goal of *Cetratus* is to ensure an uninterrupted safety integrity level of the system while high maintainability and availability is achieved.

Initially, the primary application component is only executed, for which the value of the *PRIMARY* state variable is consulted. This procedure is depicted in Figure 4.15. As observed, the *Message Router* redirects system input and output data to the corresponding application component container. To this end, the *Message Router* gets the input information from the *Provider* and delivers then to the corresponding application component container.

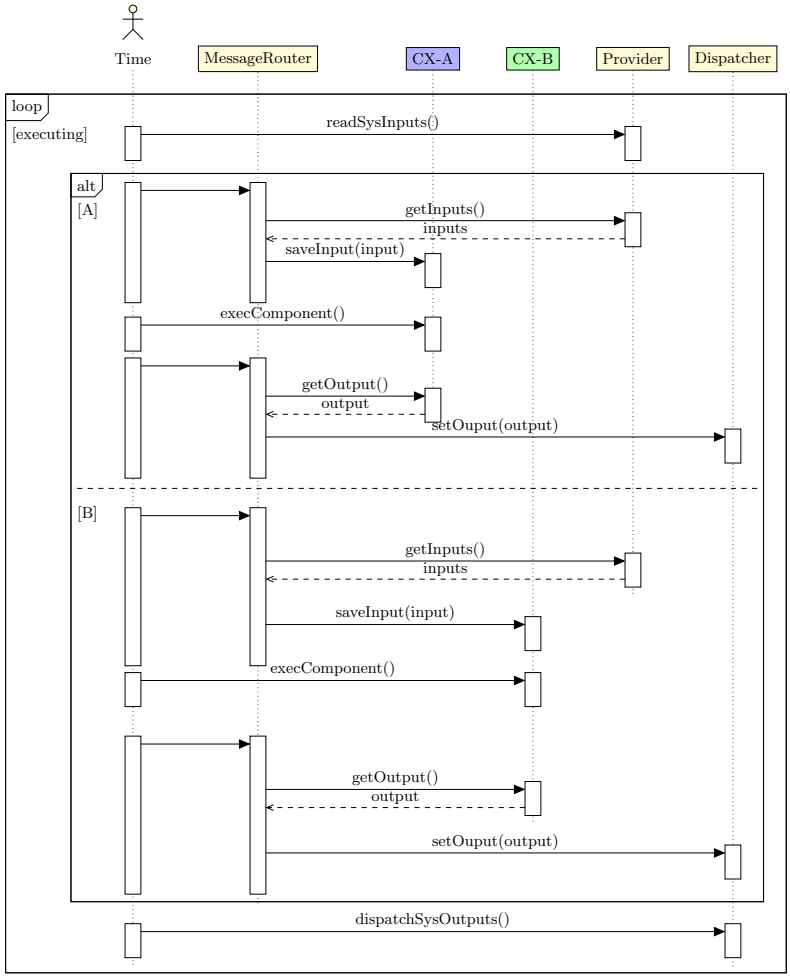


Figure 4.15: Application component execution sequence diagram

The safe live update is carried out in three main phases, which are: *Update request and patch setup* (subsection 4.2.1), *Quarantine-mode execution and monitoring* (subsection 4.2.2) and *Substitution* (subsection 4.2.3). These phases, which are represented in Figure 4.16, Figure 4.17, Figure 4.18 and Figure 4.19, are detailed in the rest of this section. A safe live update of a single application component is presented.

4.2.1 Update request & patch setup

After the dynamic patch is transferred to the target system (for example through sFTP or FTPS) and the integrity and authenticity of it are checked (R_6 in Table 3.12), the dynamic software updating process starts with an update request by the *Updater*. Nonetheless, in alignment with the IEC 62443 standard [14], the *Updater* and *Auditor* system users shall be first identified and authenticated (R_7 in Table 3.12). This prevents that the presented live patching feature is not maliciously employed by an attacker to dodge already introduced security countermeasures.

As long as the *Auditor* endorses the software update, the *Setup* stage begins, which is performed while the former application is being executed. This phase is illustrated in Figure 4.16. Both procedures shown in Figure 4.15 and 4.16 take place simultaneously, in parallel. The application component *STATUS* state variable is then set to *Setup*.

The *Updater* module firstly checks which of the containers holds the primary application component by examining the application component *PRIMARY* state variable. This variable defines which of the A and B containers is considered currently the primary one, and therefore, which container is empty and/or available as the secondary one. The dynamic patch is then instantiated and configured in the secondary container. On the one hand, code transformation is accomplished by loading the new executable code included in the dynamic patch into the secondary container. In contrast to binary rewriting and trampolines, the indirection handling technique employed in *Cetratus* makes sure that the primary executable code is not affected by the dynamic updating process.

On the other hand, for the state transformation, the state of the old application component is wrapped, and unwrapped then in the new application component. This state is later on adjusted (if needed) through the execution of state transformers. To this end, methods defined in the $IApp$ interface, shown in Figure 4.9, are employed. The *Message Router* is used to move the wrapped application component state from the old component to the new one. Once the new state is transferred, the state transformation methods are invoked by calling the $execStateTrasformer$ method defined the $IApp$ interface. By checkpointing, the state of the old application is only obtained through the $IApp$ interface. Likewise, the state transformation is also accomplished by means of this interface. The runtime framework can not directly access the internal states of the application components, enforcing safety. In-place or by use of indirection data updates would require access privileges while running.

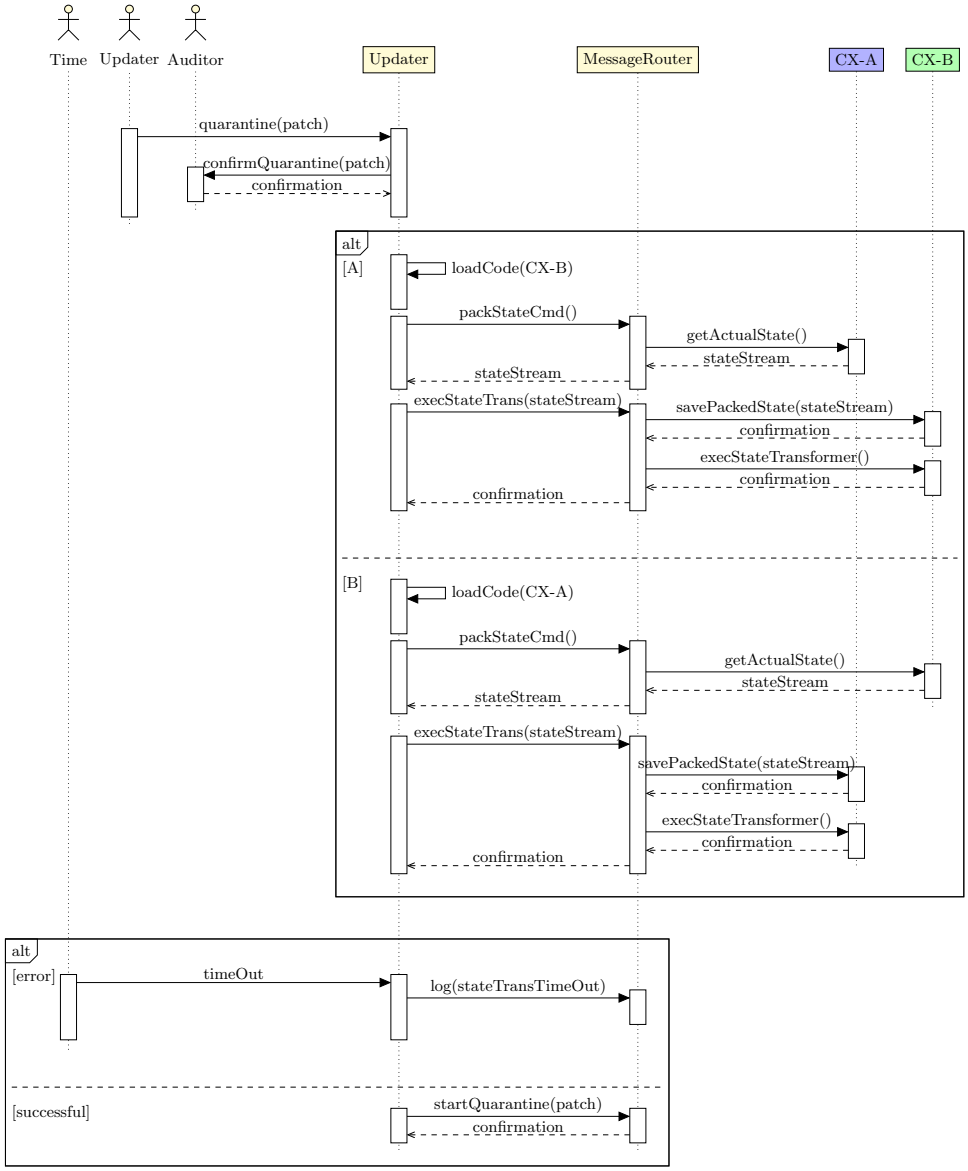


Figure 4.16: Update request & patch setup sequence diagram

As affirmed by *Wahler et al.* [133], this state has to be transformed within the same execution cycle. If this task is not completed in the same cycle, an outdated application component state could be transformed and perhaps, an inconsistent state encountered. As depicted in Figure 4.16, *Cetratus* sends a $stateTransTimeOut$ state transformation timeout notification. The dynamic update is also cancelled. The time required to perform the code and state transformations depend on the computation capabilities of the utilized hardware platform. These state transformation limitations pushes to minimize the program state size, or even better, the adoption of stateless software design patterns, as recommended by IEC 61508 [13]. In this case, simply the code transformation would be necessary. The constraints of *Cetratus* push towards the design of stateless software, as advised by safety guidelines [13]. As can be seen in Figure 4.16, the *setup* of the dynamic patch is realized while the old application component is cyclically being executed.

4.2.2 Execution & monitoring

Once the *setup* phase is successfully completed, the quarantine-mode execution and monitoring starts, where a multi-version execution is performed. At this point, *STATUS* state variable is set to *Quarantine*. Figure 4.17 shows the quarantine-mode execution procedure.

In this execution-mode, both the old application component and the new one are run in parallel, in which the input data collected by the *Provider* module is shared with both of them. In addition, the supplied input and computed output information from each application component version are sent to the *Monitor* module. However, the outcome produced by the application component enclosed in the primary container (as specified in the *PRIMARY* state variable) is only delivered by the *Message Router* to the *Dispatcher*. This procedure is iteratively accomplished in each execution cycle. As depicted in Figure 4.17, the execution of both versions is synchronized.

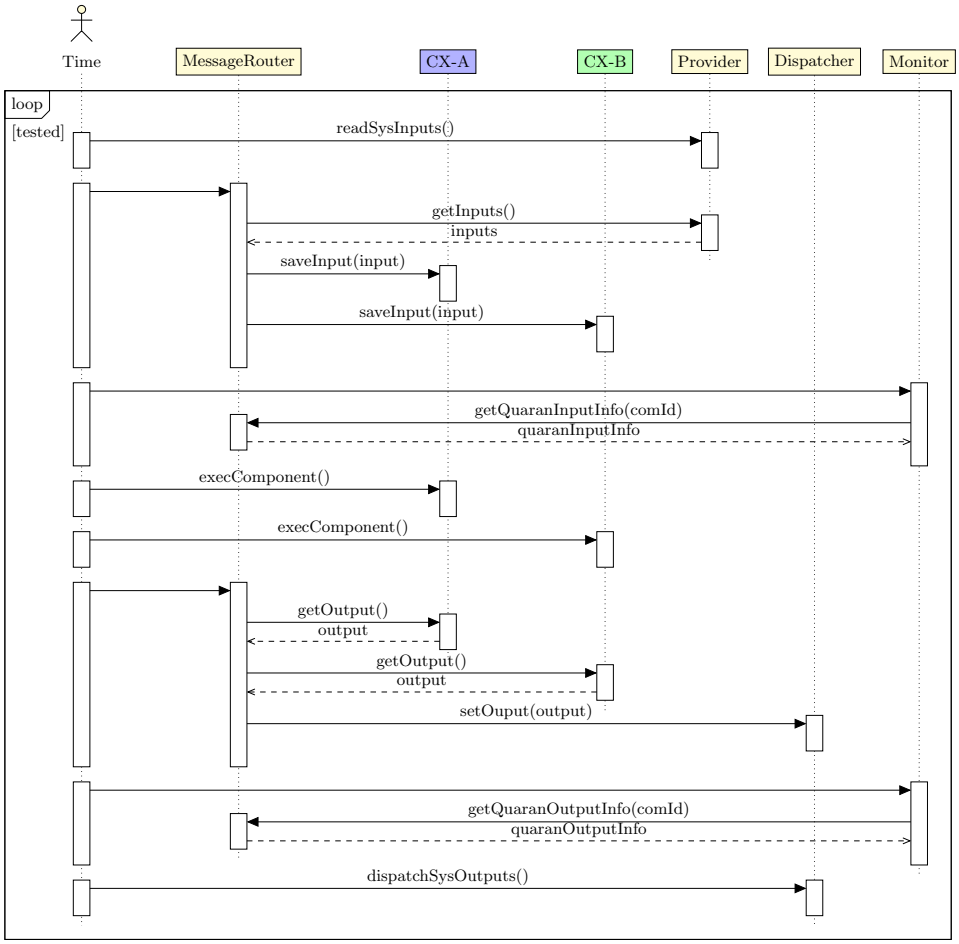


Figure 4.17: Quarantine-mode execution sequence diagram

Meanwhile, the *Monitor* module enables the quarantine-mode based monitoring for the *Auditor* system user. This process is depicted in Figure 4.18. Quarantine-mode execution information is included in the $patch\_exec\_info$ data, in which CPU and memory usage information might also be incorporated. As illustrated, this information is supplied to the *Auditor* by invoking the $getQuaranExecInfo(patch)$ method. After certain time, the *Auditor* might collect enough evidences that confirm the correctness of the new application component and validate the dynamic patch. The given patch is then marked as validated. The *STATUS* state variable is set to $Validated$.

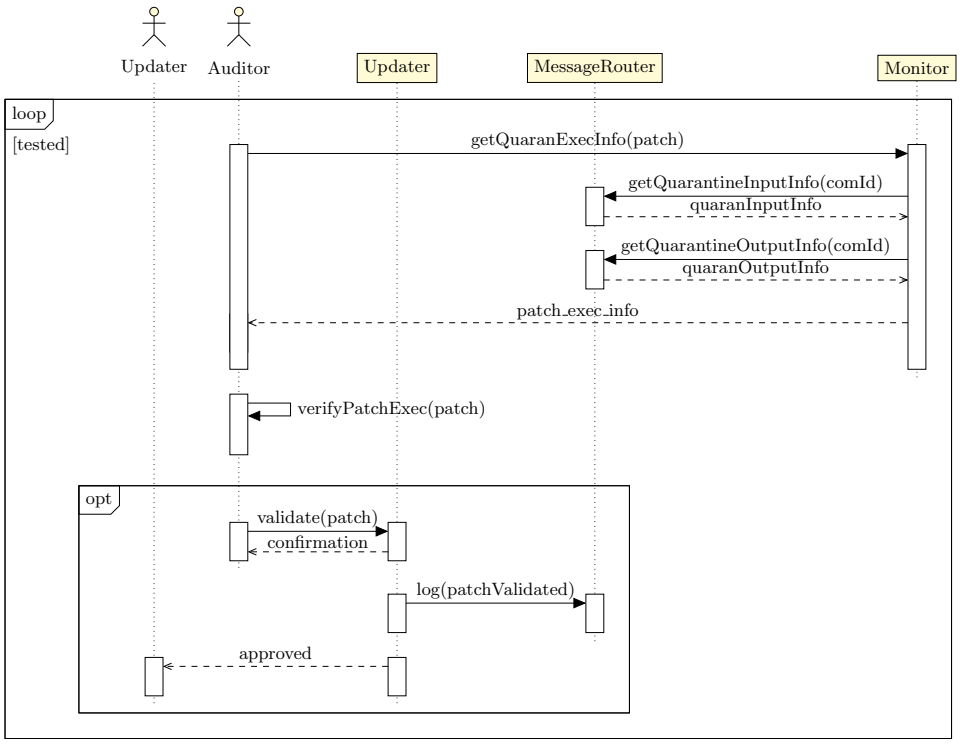


Figure 4.18: Quarantine-mode monitoring sequence diagram

4.2.3 Substitution

In the last phase of the dynamic software update process, the substitution operation is performed. To that end, the dynamic patch needs to previously be validated by the *Auditor* (the value of the *STATUS* state variable has to be $Validated$). A confirmation from this system user is also necessary later on. Alternatively, the software update can be discarded at any moment by the *Updater* or the *Auditor*. In the substitution phase, the quarantine-mode execution and monitoring is stopped first. Afterwards, if the patch is removed, the quarantine-mode setup is disabled. The old application

component is not replaced with the new one (the *PRIMARY* state variable is not commutated). On the contrary, if the software update is endorsed by the *Auditor*, the substitution of the application component is accomplished and the *PRIMARY* state variable, which indicates to the system the version of the application component to be executed, is switched. Figure 4.19 depicts this procedure.

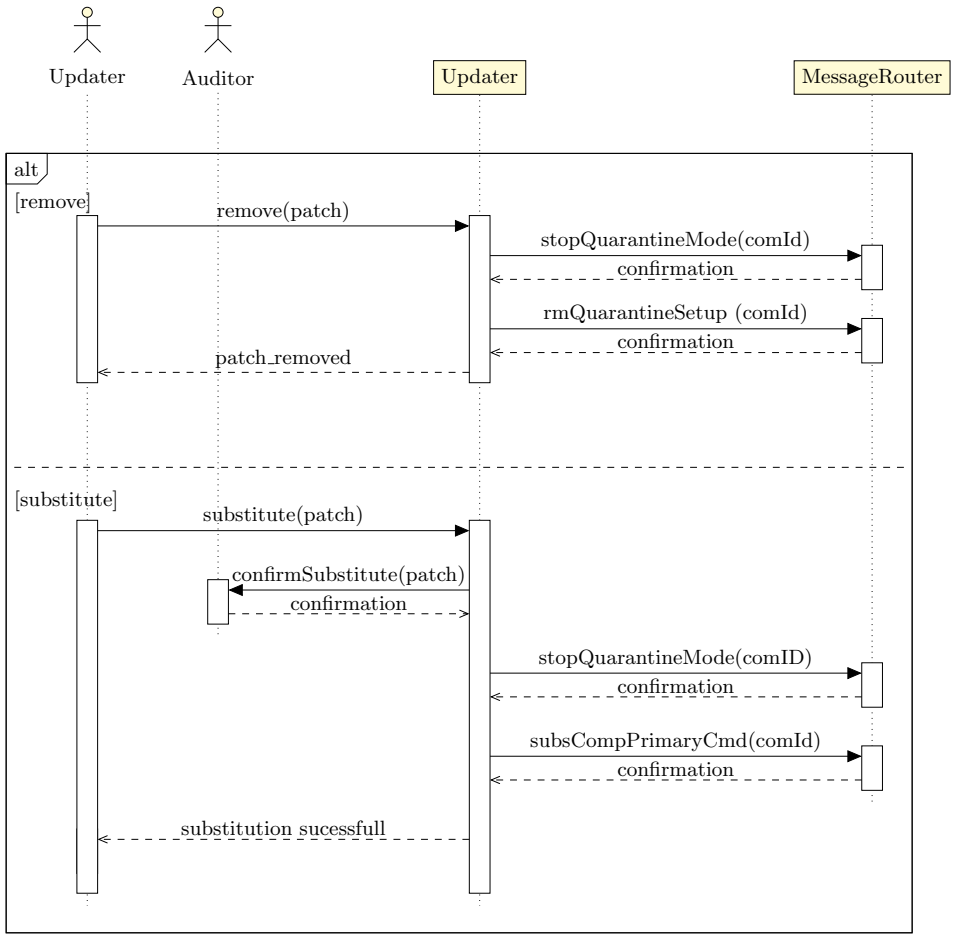


Figure 4.19: Substitution sequence diagram

4.3 Implementation

Safety is a big concern in dynamic software updates. Even more, in our case, the proposed solution might be certified against safety standards, such as IEC 61508 [13], and be adopted on the design, development and maintenance of a safety-critical system. Thus, not only software safety, but whole system safety has to be also warranted. A prototype version of *Cetratus* has been implemented as an Ada runtime system, which fulfils with the software architecture and design level safety requirements and provides a proof-of-concept of the proposed solution.

As claimed by *Stoyle* [94], the most important technical challenge resides on how to address the unsafe features of the C programming language. Instead of C, TAL and Popcorn were used by *Hicks* [75]. TAL incorporates typing rules, which guarantee type safety, control flow safety and memory safety on an untyped assembly language. A variant of TAL was developed and employed by *Hicks* for Popcorn, a safe C-like language. In this programming language, unsafe C features, for instance pointer casts or generic address operators, are excluded. Static analyses are often also performed to ensure the correctness of software patches when unsafe programming languages are used. This is the main reason why the Ada programming language has been chosen for the implementation of the *Cetratus* prototype system. Besides, this language is widely adopted on the development of safety-critical and/or mission-critical systems and it is highly advised by the safety community [13].

In a similar way to *Wahler et al.* [133], utilities from an underlying commercial operating system are employed, since dynamic software updates are indirectly supported by these operating systems. Long-term support and liability is also achieved. In this case, the implemented prototype of *Cetratus* is POSIX compatible. It has initially been integrated with Real-Time Linux, running on a x86 architecture computer (Intel I5 core running at 1.2 GHz). Even that predictability and efficiency is decreased, as shown by *De la Puente et al.* [145], POSIX interface provides high portability of the solution among operating systems, and hence architectures. Indeed, the implemented *Cetratus* proof-of-concept could be reasonably cross-compiled for another computer architecture, such as ARM or PowerPC.

In order to be able to accomplish a dynamic software update, it is advised to reach recurrently a possible update point. For this reason, a cyclic behaviour of the application is required, which means that application components should run in an infinite loop. This software design pattern is highly recommended by the safety software guidelines [13]. A time-triggered architecture is employed. In contrast to the usual dynamic software updating process, in *Cetratus*, there is no need for any update point for the initial-

ization of the patch. Nevertheless, an update point has to be reached to start the quarantine-mode *execution and monitoring* and the *substitution* stages. Furthermore, similarly to the approach presented by *P. Hosek* and *C. Cadar* [80], the thread execution shall be synchronized in the quarantine-mode execution, where the old and the new program run in parallel.

Figure 4.20 depicts the cyclic execution timeline of all *Cetratus* framework modules and application components depicted in Figure 4.3. In order to ensure that the real-time requirements are satisfied, at design time, execution time is statically allocated for all framework elements, including both A and B containers for each application component (R_4 in Table 3.12).

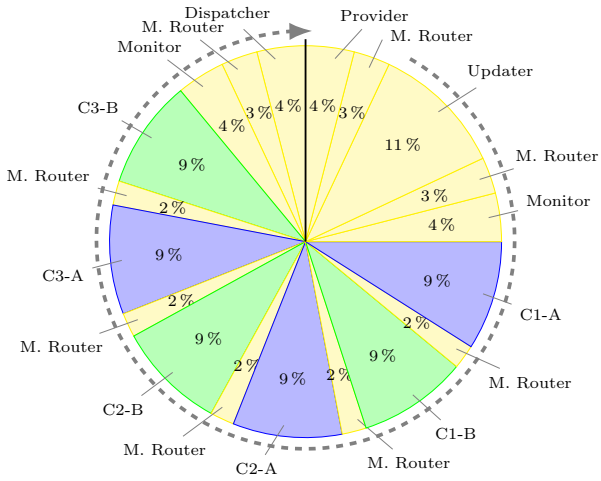


Figure 4.20: Schedule of *Cetratus* runtime modules and application components

As observed in Figure 4.20, the DSU functionality demands higher platform resources. Double memory and execution time is needed for each application component (two containers are created for each one). Some execution time has to be also allocated for the dynamic software updating service (enabled by the *Updater* and *Monitor* modules). Nevertheless, in case a multi-core platform is employed instead of a single-core one, another processor could be used to execute the quarantine-mode container. This would decrease the required execution time in each processor.

4.4 Conclusions

In this chapter, the *Cetratus* framework has been described. As stated throughout, the proposed solution fulfils with the main requirements presented in Table 3.12 for enabling dynamic updates in safety-critical systems.

In order to protect the computer program, and in consequence the system, a quarantine-mode execution and monitoring mode is adopted (R_1 in Table 3.12), where the new software version is executed and monitored for software and patching faults (R_2 in Table 3.12). This mode allows the internal testing of new application component versions (R_3 in Table 3.12). To this end, two different containers are created for each upgradable application component. These containers are statically allocated, as recommended by the safety standards [13] (R_3 in Table 3.12). A static temporal partitioning is also implemented (R_4 in Table 3.12), ensuring that application executions of a container do not compromise the timing properties of others.

Moreover, an authentication and authorization of system users is enforced (R_7 in Table 3.12). This measure prevents that the dynamic updating feature is not maliciously employed by an attacker to dodge currently implemented security countermeasures. At the update request, the digital signature of the dynamic patch is also verified to ensure its integrity and authenticity (R_6 in Table 3.12). Table 4.2 summarizes the employed methods to satisfy the requirements defined in Table 3.12.

ID	Definition	Measure
R_1	Protection against patching failures	A quarantine-mode execution and monitoring mode, similar to sandboxing. For this purpose, partitioning techniques are used
R_2	Software updates fault monitoring	The new software is executed and monitored in such mode for software and patching failures
R_3	Static resource allocation	Upgradable application component containers are statically allocated
R_4	Time-triggered architecture	Temporal partitions are initially defined
R_5	Internal testing capability	The new software version is tested while quarantine-mode execution and monitoring phase
R_6	Software patch integrity and confidentiality	A digital signature is provided together with the dynamic patch.
R_7	Authentication and authorization of system users	The *Updater* and *Auditor* system users are authenticated.

Table 4.2: Employed methods for safe and secure DSU compliance

5 Validation

In this chapter the validation of the proposed solution is presented. The validation has been carried out by means of an initial validation and two case studies:

- Initial Validation: The proposed solution is initially validated to ensure that the developed prototype fulfils the functional requirements.
- Smart Energy Case Study: A smart building energy management application is considered. In this case study, an additional security layer is dynamically incorporated to enhance customer data security and privacy.
- Railway Case Study: In the railway case study, the *Euroradio* component, which is responsible for safety-related digital communications between the train and track-side equipment is upgraded. A more secure MAC scheme is incorporated to the system while runtime.

5.1 Initial Validation

For the initial validation, the proposed solution is evaluated through an experiment. First of all, the system reads several inputs. After that, the system output is determined based on the obtained data. This procedure is cyclically executed every 20 ms. For the assessment of the dynamic software updating functionality presented in this paper, an additional sawtooth wave signal is provided to the system as an input. This signal is then transformed into a triangular wave by the program and settled as an extra system output. Through the dynamic update, the signal processing algorithm is adjusted, smoothing the output signal. Instead of the triangular wave signal, a parabola is produced. The described process is shown in Figure 5.1.

As shown in Figure 5.1, a triangular wave signal is initially produced by the system. After a dynamic update request from the Updater, the new software patch is initialized. The quarantine-mode execution and monitoring step would indefinitely be then accomplished until a remove or substitution command is received from the Updater. For this purpose, in this experiment, the behaviour of the *Updater* and the *Auditor* were predefined beforehand. At cycle 63, the software patch is configured and initialized.

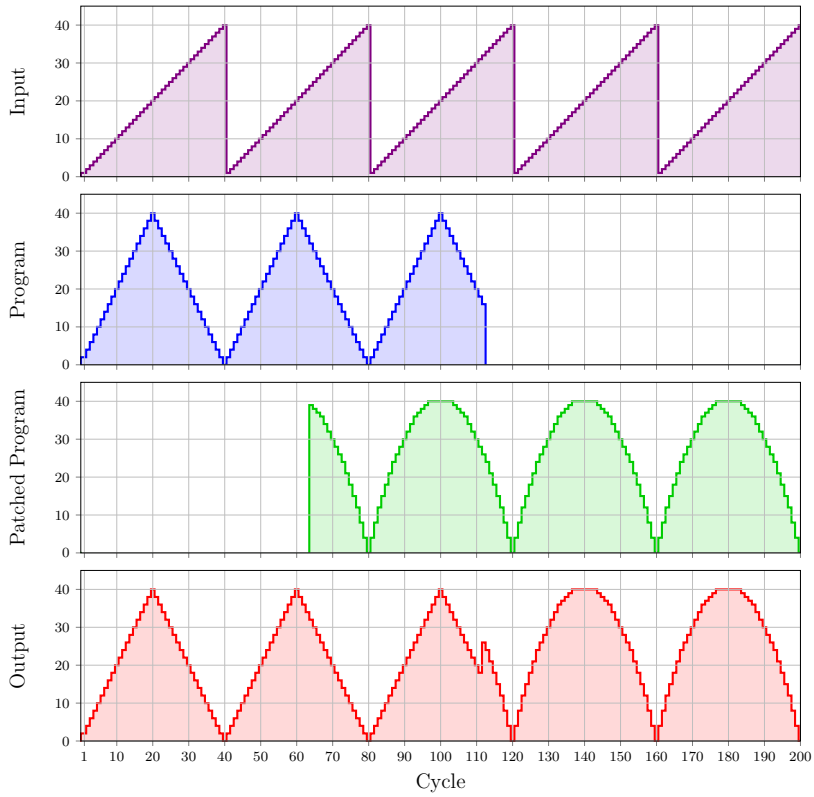


Figure 5.1: Validation - Input, output & internal signals

The new software patch is then internally executed and monitored between cycles 64 and 112. At this point, a substitution is finally performed. The old *Program* application component is halted. This is the reason why no output computed by the old *Program* is no longer illustrated after cycle 112 in Figure 5.1.

Figure 5.2 shows the CPU usage on each execution cycle for the implemented Cetratus runtime, the old program and the new software components. As observed, the implemented runtime introduces a relatively small overhead. Nevertheless, a meaningful boost is appreciated at cycle 63, where the *setup* of the patch is accomplished, in which the code and state transformations are performed. The required amount of time is determined by the size of new executable code to be imported (including state transformation functions), the size of the application component state to be transferred, and the computational capabilities of the hardware platform. A time-out notification is sent by Cetratus in case the state was not possible to be transferred within a single execution cycle. The dynamic software update process is also aborted.

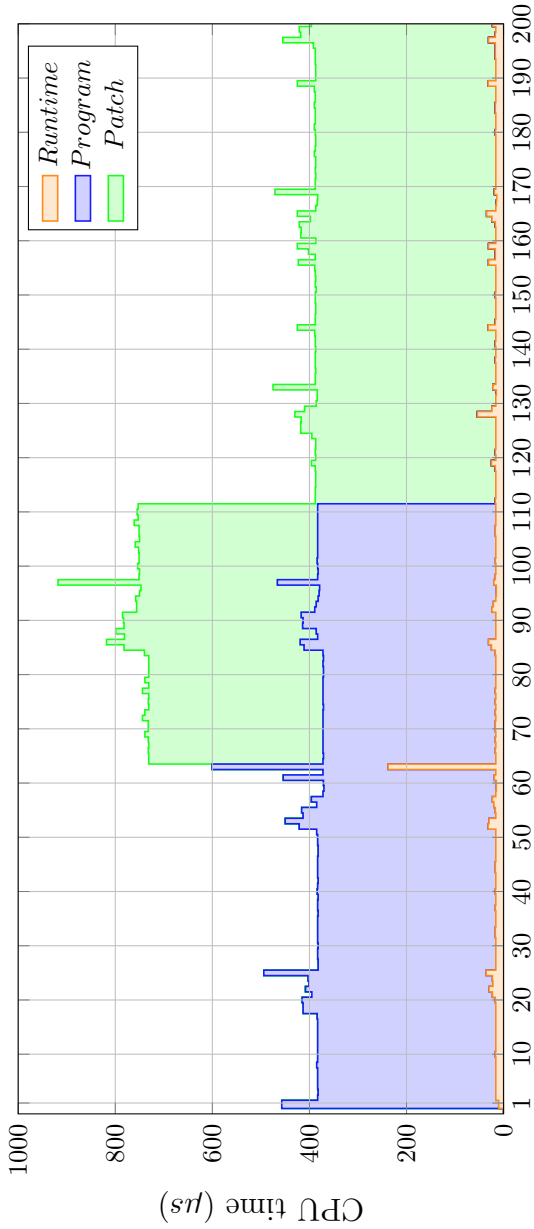


Figure 5.2: Validation - CPU usage

5.2 Smart Energy Case Study

These days, information and communication technologies are being employed and integrated towards the accomplishment of interconnected and intelligent smart cities, where the comprehensive goal is to improve the quality of living of citizens. This might include the reduction of wastage, resource consumption and/or the improvement of overall living costs. To this aim, smart and advanced services are offered. High interconnectivity among all sensing, storing, processing and analysing devices is fundamental, a tendency enabled and promoted by the IoT and the IIoT technologies.

In case of energy sector, *Smart Energy* and *Smart Energy Systems* refer to the design and implementation of sustainable and cost-effective energy management strategies [146]. This topic is actually being analysed and investigated in several research projects, such as in/by CITyFiED [147]. The goal of this project is "to develop a replicable, systemic and integrated strategy to adapt European cities and urban ecosystems into the smart city of the future, focusing on reducing the energy demand and Green House Gas (GHG) emissions and increasing the use of renewable energy sources by developing and implementing innovative technologies and methodologies for building renovation, smart grid and district heating networks and their interfaces with ICTs and Mobility" [147].

5.2.1 Building Energy Management System

A smart grid platform which makes use of information and communication technologies is employed in CITyFiED for grid management solutions. Figure 5.3 illustrates the adopted approach. The system is divided in different levels. First, a Building Energy Management System (BEMS) is defined, which gathers the energy flows information of each of the buildings in the district. Electric car charging points are also installed in some parking slots. Secondly, the District Energy Management System (DEMS) monitors energy generation and distribution at district level. Data collected by the DEMS and BEMS system is transmitted to the application server. For this purpose, third party information, e.g. weather forecast or electricity tariff published by the Energy Service Company (ESCO) is used. Moreover, a remote HMI provides home actual status, historical data and trending, as well as third-party information. The graphical user interface will provide a dashboard to depict such data, as well as alerts and notification in case of unnecessary waste of energy.

In this case study, a smart building electrical energy management application is considered, consisting of the BEMS, and a Building Energy Optimization Service (BEOS) cloud application. On the one hand, the BEMS is responsible of monitoring and controlling diverse energy-related facilities

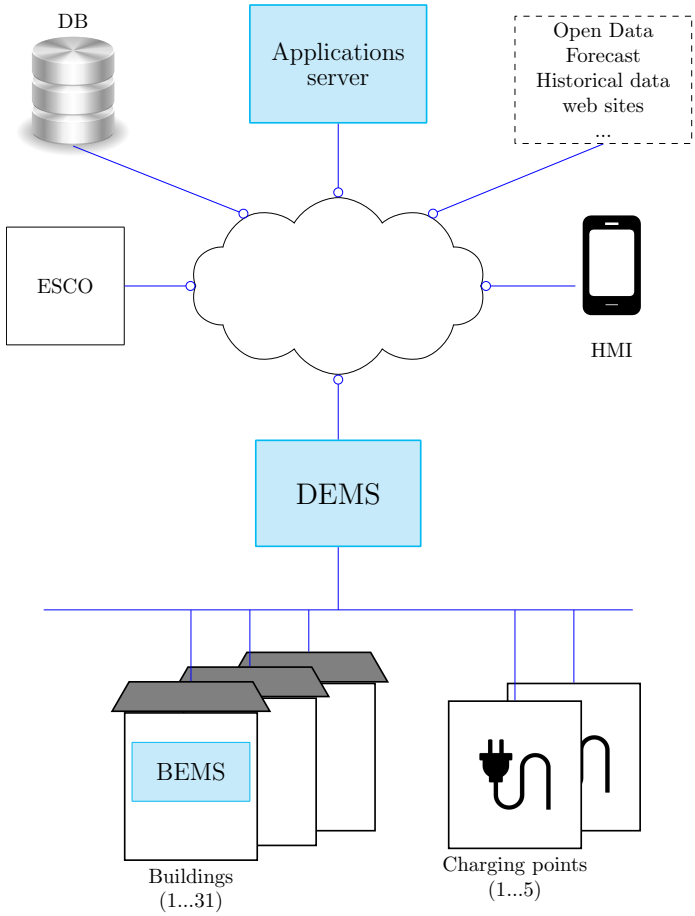


Figure 5.3: Smart Grid and ICT system in CITyFiED [147]

on a residential building. Firstly, different energy sources are managed and scheduled: a wind turbine, solar cells and the electrical grid. It is assumed that a wind turbine and solar cells have been installed at the roof of the building. Secondly, the supplied electric power is used by various home appliances, the elevator and a electric car charger. The BEMS continuously measures the energy consumption of these equipments. Finally, an energy storage unit is also used. The BEMS directly controls the wind turbine and solar cells, as well as the energy storage unit and the electric car charger, which have safety requirements. We assume that the electric car charging points are installed in an underground parking garage, within the building.

Figure 5.4 depicts the described smart energy application. On the contrary to the Smart Grid and ICT system proposed in CITyFiED [147], the DEMS is ignored.

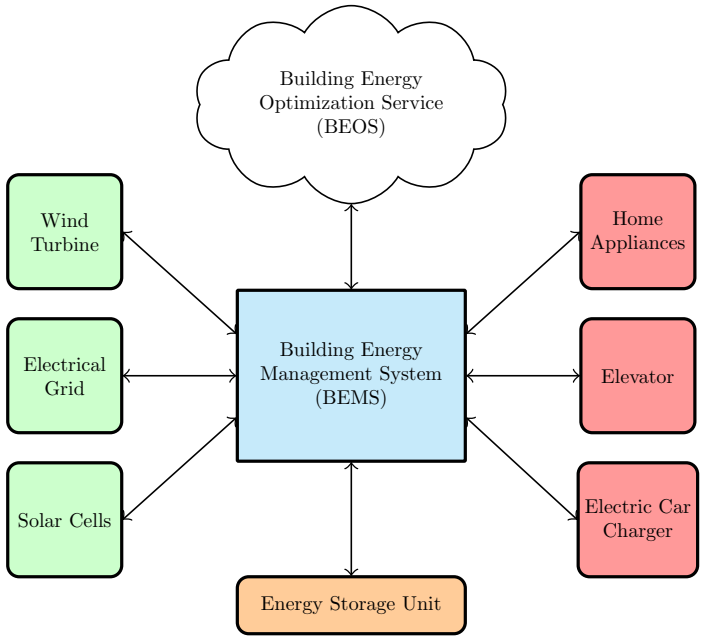


Figure 5.4: Building Energy Management System (BEMS)

On the other hand, all energy consumption and savings measurements are transmitted to a BEOS. This cloud application estimates and optimizes the overall building energy consumption for higher energy efficiency and cost reduction. For this purpose, in addition to the data sent by the BEMS, other information sources are analysed: the actual and expected electricity fees, and weather forecasts for renewable energy sources estimations.

In this chapter, a mixed-criticality software architecture based on the *Cetratus* runtime framework is proposed for the BEMS. The presented scheme allows the dynamic update of application components. A live update example is also later on provided, where the customer data privacy is improved through the use of homomorphic cryptography techniques. Certainly, the cloud service should be able or be adapted to process this kind of encrypted data.

5.2.2 Software Architecture

The software architecture of the BEMS is shown in Figure 5.5. The application software running in the BEMS is divided among several application components, identified in Table 5.1. Wind and solar energy productions are measured by the C-WEM and C-SEM components. The C-HEM application component measures the electrical energy consumption of home appliances, including non-shift habits, such as lighting, installed in each

flat. The C-EEM monitors the energy consumption of the elevator. The C-ESC controller manages the energy storage unit, where previously produced and captured energy is accumulated for use at a later time. For this purpose, rechargeable batteries are employed. The C-ECC manages and controls the electric vehicle charging station. Due to the involved risks, such as electrical surges and leakages, both the C-ECC and C-ESC components need to fulfil with safety standards.

ID	Name
C-BEM	Building Energy Manager
C-SDC	Secure Data Collector
C-WEM	Wind Energy Meter
C-SEM	Solar Energy Meter
C-HEM	Home Energy Meter
C-EEM	Elevator Energy Meter
C-ECC	Electric Charger Controller
C-ESC	Energy Storage Controller

Table 5.1: Application components in BEMS

The C-BEM is the overall building energy manager, and it is able to decide when the electrical energy is purchased and obtained from the grid. These electrical energy purchasing and saving profiles might manually be determined or, preferably, requested to the BEOS. All energy production and consumption data are gathered by the C-SDC. This component shall integrate the required security countermeasures to ensure the confidentiality and integrity of the records sent to the cloud application.

Two containers (depicted in blue and green color in Figure 5.5), which provide isolated execution environments both in the spatial and temporal domain, are defined for each of the application components (except for the C-ECC and C-ESC). These containers are defined in the software design phase, and are statically allocated, as advised by safety guidelines [13]. Safety-related application components C-ECC and C-ESC (depicted in red color in Figure 5.5) are determined as not upgradable, since in case that safety hazards and risks have been properly addressed or the operational conditions of the system do not change, software updates are not recommended in safety engineering [13]. Therefore, a secondary container is not required.

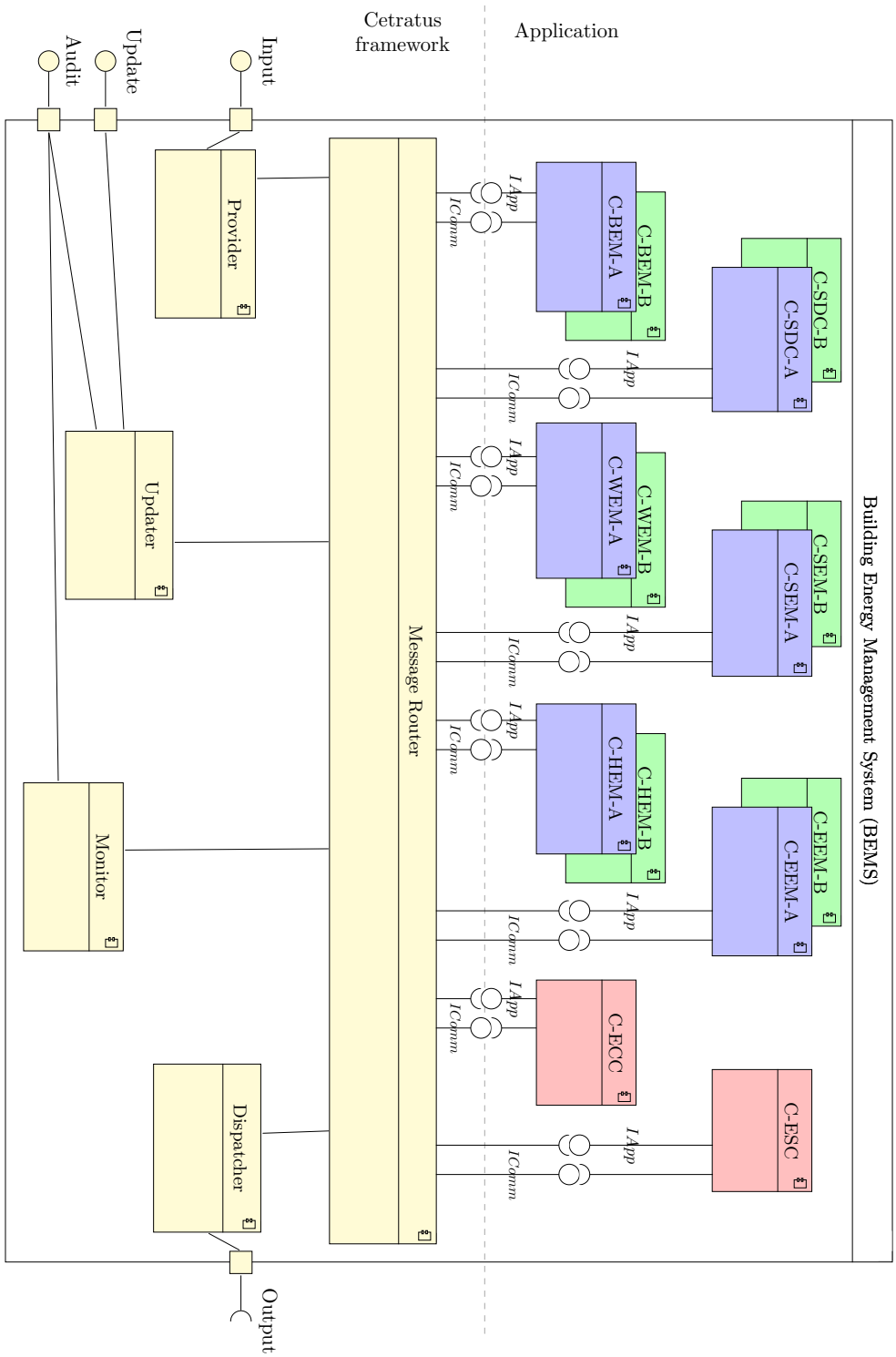


Figure 5.5: Software architecture of the Building Energy Management System

As far as *Cetratus* framework components are concerned, the dynamic software update functionality is enabled by the *Updater* and *Monitor* runtime modules. On the one hand, the dynamic software updating process is managed by the *Updater*, which performs the required code and state transformations. All application components shall be compliant with the *IApp* interface and be developed under the proposed framework. On the other hand, patch execution monitoring data is gathered by the *Auditor*. The execution footprint, such as memory usage, CPU or timing behaviour, is also gathered.

Input and output abstraction are offered by the *Provider* and *Dispatcher* framework modules, which act as wrappers to the underlying specific input and output drivers, such as fieldbus communications, digital or analogous I/O, etc. This information is forwarded to the corresponding application components through the *Message Router*. Message passing procedure is transparently handled by the *Message Router*, without modifying the data contained in the message.

Finally, the *Message Router* enables the inter-component and other system module communications, by means of a message-passing mechanism. The *IComm* interface is offered to the application components for such inter-component communications. This interface defines the methods to create, open and close a communication channel from which messages are sent and received.

5.2.3 Partitioning

Typically there are many agents involved in the design of such complex systems, and their integration is a growing concern. In order to assure that specification, design, implementation and certification (if needed) stages are independent among components, partitioning is used. A partition is a strictly independent execution environment that is protected from other partitions. For this purpose, independence of execution both in the temporal and spatial domains shall be achieved.

On the one hand, temporal partitioning ensures that application executions of a partition do not compromise the timing properties of other partitions by monopolizing the CPU or shutting down the system, for instance. To achieve this, applications are executed only during the time slices they are assigned to. During this time, services received from shared resources must not be affected by applications in other partitions. In the case of control applications as BEMS, it is essential to guarantee that each temporal partition is assigned enough processing time to complete its execution. On the other hand, spatial partitioning ensures that the software within a partition can not access memory resources of another partition. To

this end, the access to memory regions where data and code reside is controlled, which avoids unauthorised read/write operations and commanding resources hosted in different partitions.

Two main partitioning approaches exist: hypervisors and partitioning enabled operating systems. In the case of hypervisors, e.g. *Xtratum* [50] [51], different operating systems can be run in a processing element, creating completely isolated virtual execution environments. Regarding partitioning enabled operating systems, isolation is obtained by enhancing the host operating system's features so that partitioning techniques can be implemented. As an example of this approach, the *INTEGRITY* RTOS (POSIX compliant) developed by *Green Hills Software* has been certified for security, safety and reliability domains, including IEC 61508 SIL3 [13]. Spatial partitioning is obtained by Virtual Address Space (VAR)s, which are protected memory regions of code and data that can only be reached by authorised processes. A *Partition Scheduler* is used to set a cyclic schedule of temporal partitions where different VARs are allocated, so that they can be bounded in time by designers. The combination of these two elements provides a great flexibility for virtualization, since code and data stored at a certain VAR can be executed several times in different temporal partitions if desired, without violating any partitioning principle.

Each component of the BEMS application is allocated in an independent spatial partition to have isolation. Each one of these spatial partitions is executed at least in one temporal partition, but may be executed in more than one depending on the containing component, such as *Message Router* or *Monitor* components, which are executed several times.

Figure 5.6 shows a proposal for scheduling the execution of the partitions. The *Major Frame* is the execution that is repeated periodically and it is defined at the design phase using adequate timing analysis techniques. Timing analysis techniques are mathematical methods to formally calculate the response time of a system, easing its design towards obtaining the certainty that the system is schedulable even in the worst-case scenario [148]. If a component is going to be updated, a temporal partition for the execution of its secondary version must be scheduled, as shown in the picture. Moreover, extra time within the provider-dispatcher frame can be allocated in case another component is going to be executed in the future, considering its secondary version for dynamic updating as well (components C-X-A and C-X-B). This provides the system a higher degree of expandability without compromising its temporal restrictions.

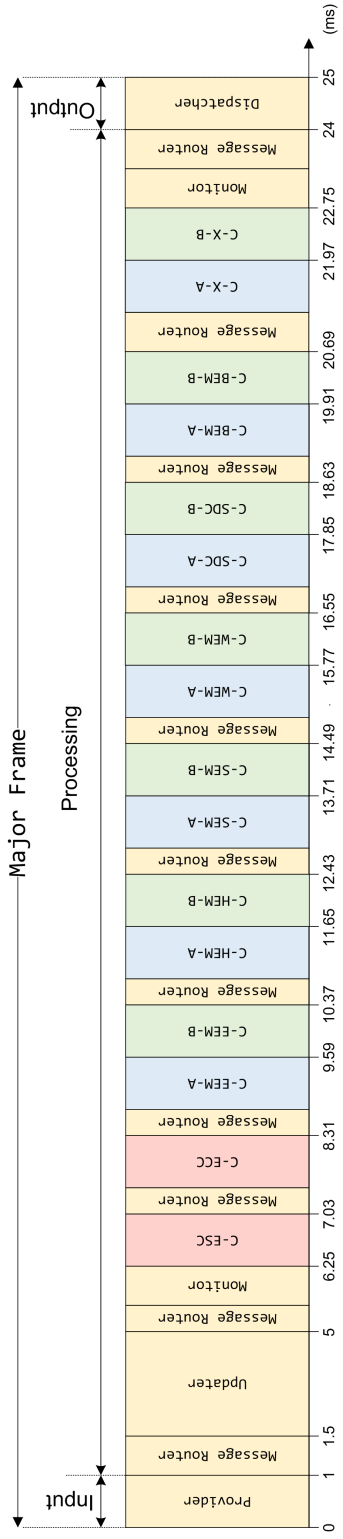


Figure 5.6: Temporal partition scheduling

As shown in Figure 5.6, the *Major Frame* is divided into three stages: First one acquires system inputs and delivers them among their corresponding components. Then, each component performs its processing tasks during their assigned periods of time, and finally the last stage corresponds to system output delivery. The *Provider* component is executed first, since it is the only one that has an input interface. As shown in the software architecture depicted in Figure 5.5, it is connected to the *Message Router* so that inputs can be delivered to the rest of the components. The *Message Router* component allows communication between all components, which is why it is executed after each component has been run. *Monitor* and *Updater* components are in charge of controlling the software update process, and finally the *Dispatcher*, using the information transmitted through the *Message Router*, selects the outputs from the different application components. Thanks to partitioning, it is guaranteed that when running components in *Cetratus*, if any misfunction occurs during the dynamic software update process, it shall be contained and it will not jeopardize the correct functioning of the rest of the system.

In the *INTEGRITY* RTOS, the temporal partitions are defined by the following parameters:

- *AddressSpace*: allows allocating spatial partitions within temporal partitions.
- *Offset*: sets the relative time in the *Major Frame* when the partition starts its execution.
- *Exectime*: sets the length of time it is executed. It is guaranteed that after this time, no matter what happens, the execution of that partition will be stopped and the scheduled next one will start.

Therefore, the *Major Frame* will be the sum of all temporal partitions in the schedule. In this case, it has been set to 25 *ms*, which has been proved to meet all temporal constraints of the system. In Listing 5.1 an extract from the *INTEGRITY Integration File* is shown. Here the scheduling of the temporal partitions within the periodic the *Major Frame* is defined. Temporal partitions must be, at least, long enough to allow components allocated to them to be executed in their worst-case execution times.

```
PartitionSchedule BEMS

  MajorFramePeriod 25
  # Major Frame period is 25 millisecond long

  Partition Provider
    AddressSpace Provider #Provider VAS
    Offset 0
    Exectime 1
    # At 0 miliseconds into the major frame run 1 miliseconds
  EndPartition
...
  Partition Updater
    AddressSpace Updater  #Updater VAS
    Offset 1.5
    Exectime 3.5
  EndPartition

  Partition Message_Router
    AddressSpace Message_Router
    Offset 5
    Exectime 0.5
  EndPartition

  Partition Monitor
    AddressSpace Monitor
    Offset 5.5
    Exectime 0.75
  EndPartition
...
  Partition C-SDC_A
    AddressSpace CSDC_A #Secure Data Collector_A VAS
    Offset 17.07
    Exectime 0.78
  EndPartition

  Partition C-SDC_B
    AddressSpace CSDC_B #Secure Data Collector_B VAS
    Offset 17.85
    Exectime 0.78
  EndPartition
...
  Partition Dispatcher
    AddressSpace Dispatcher #Dispatcher VAS
    Offset 24
    Exectime 1
  EndPartition

EndPartitionSchedule
```

Listing 5.1: *Partition Scheduler* integration file

5.2.4 Live Patching

In this section, a live update example is presented, where a new security layer is incorporated to enhance customer data security and privacy. Concretely, the C-SDC application component is upgraded. In this new application component, a homomorphic encryption algorithm is integrated [34]. Through homomorphic encryption, all data exchanged by the BEMS with third-party cloud services is then protected against information leakages. Other security weaknesses, bugs or misconfigurations could also be fixed through this update. A prototype of the presented BEMS has been developed, which is executed on a x86 industrial computer. The implemented *Cetratus* runtime framework has been integrated over *INTEGRITY* RTOS. Regarding DSU, features from the underlying operating system are employed.

Although a secure communication channel is used for the transmission of energy production, savings and consumption data, e.g. by means of TLS or any other encrypted and authenticated communication protocol, the smart energy data is stored and processed in clear text by third-party services. Confidential information is then exposed to them. As stated previously, software updates are necessary to address security and privacy issues that might be encountered during the operational period of the system. Assuming that the system has already been deployed and it is being executed, a live update would be necessary to address this problem. As a solution to the presented privacy issue, homomorphic cryptography algorithms might be employed [149].

Figure 5.7 illustrates the outputs produced processed by both C-SDC application component, and the data transmitted to the BEOS cloud service by the *Dispatcher*, during the quarantine-mode based live patching procedure. The new C-SDC also encrypts such data in that processing phase. As depicted, at the beginning, plain data gathered by the first component version is transmitted to the cloud application. The BEOS receives, stores and processes plain customer energy production, savings and consumption data, where customer living behaviour patterns and private information might be obtained from. At 26^{th} hour, the second version of the C-SDC application component is initialized. This component is then internally executed and monitored on quarantine-mode. During this stage, both versions are executed and the behaviour of the new component verified.

After the validation, a substitution of the former application component is performed. This step is accomplished at hour 63. The former C-SDC component is then halted. As shown, after performing the live update, customer information is hidden. The encrypted data computed by the second component version is sent to the BEOS. Citizen privacy is then ensured.

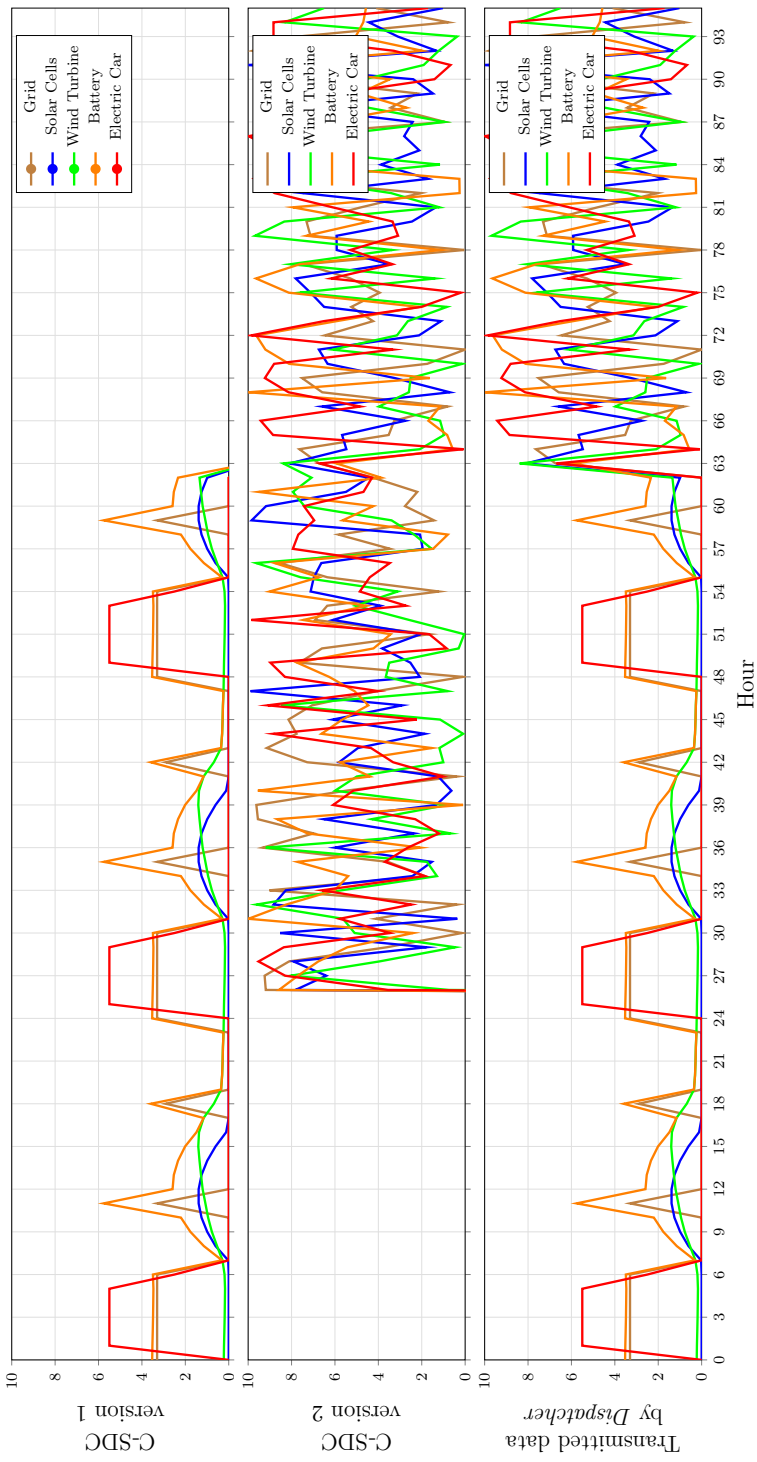


Figure 5.7: Energy production, savings and consumption data (in kWh) computed by both C-SDC application component versions and transmitted information to the cloud service

Figure 5.8 shows the system performance in terms of CPU and response times. At the top, the CPU time of both C-SDC version application component versions during the live update process is depicted in μs. As said before, The first version C-SDC component is executed for the first 26 hours, and after that, when its secondary version starts its execution, a notable increase in CPU time is observable. This period of time, which is in fact the quarantine-mode period, the use of CPU will reach its peak, since both components are being executed at the same time. Almost the whole available CPU time is required at this stage for both application components. After that, when the quarantine-mode and live update process are already accomplished, the usage decreases again to a slightly higher value than the one at the beginning. The new C-SDC application component version demands higher CPU usage than the old one due to the higher computational cost required for the data encryption operations. Consequently, the new application component makes use of all the available CPU time assigned to it.

Temporal requirements of the system might be in danger when the total CPU usage demanded by the application components and/or other system modules increases significantly. The plot at the bottom of Figure 5.8 shows the system response time, i.e. the time required by the application to produce and send the output. As noted, the system response time values do not go beyond the time limits defined through temporal partitions. As stated before, these temporal partitions have been designed so that all temporal requirements can be met in any case. The system shall be able to deliver outputs before the end of the major frame period (25 ms). The system is then capable of ensuring all the temporal requirements and constraints while both application component versions are being executed.

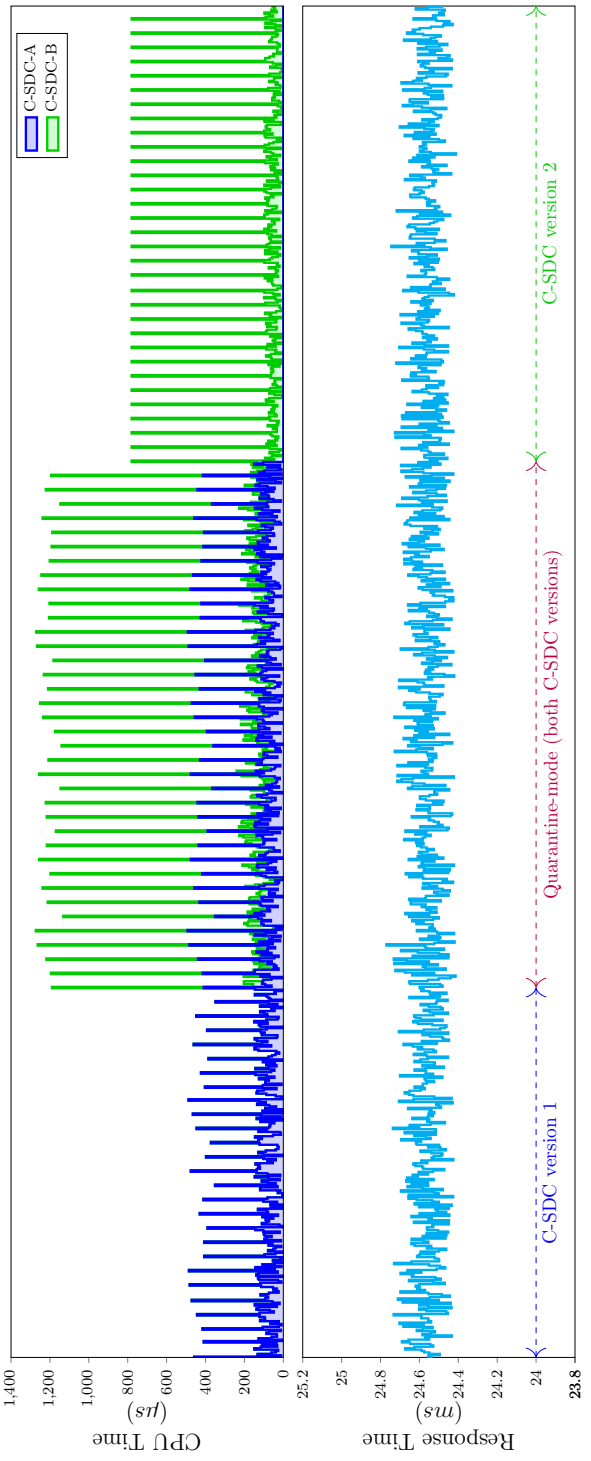


Figure 5.8: CPU and system response times during live update (*INTEGRITY* RTOS, executed on a x86 industrial computer)

5.3 Railway Case Study

In this section, a railway case study is presented. Even though it does not consider a strictly zero downtime safety-critical application (an usual software update could be accomplished at the end of the mission for example), a theoretical interest exists.

5.3.1 European Railway Traffic Management System

The European Railway Traffic Management System (ERTMS) is an European major project which aims at promoting interoperability and safety by replacing previous national signalling systems with an unique signalling and communication standard. It is, at the time of writing, formed upon two parts: The Global System for Mobile Communications - Railway (GSM-R) communications and the European Train Control System (ETCS). The GSM-R is a radio system which provides voice and data communication between the track and the train. On the contrary, the ETCS system is an Automatic Train Protection (ATP) system, which constantly computes a safe maximum speed for the train. For safety purposes, the on-board could then gain control over the driver in case the authorised speed is exceeded.

Three levels of operation modes are defined in ERTMS. These levels are:

- **Level 1**: In the first level, a non-continuous communication between the train and track-side equipment exists. This is usually achieved by Euro-balises. Furthermore, the train position is detected by former track-side systems. Traditional line-side are also needed.
- **Level 2**: In this level, GSM-R technology is incorporated for a continuous communication and supervision by and for the train. Likewise level 1, the detection of the train is accomplished by equipments outside ERTMS. Former line-side are not compulsory.
- **Level 3**: Lastly, the third level involves full ERTMS-based train integrity, supervision and management, which is entirely based on GSM-R radio communications. Line-side signals and track-side equipment except Euro-balises are not longer necessary.

Figure 5.9 shows the high-level picture of the ERTMS railway signalling system for levels 2 and 3. The European Vital Computer (EVC), also denoted Eurocab, is the heart of the ETCS system. This safety-related controller supervises the speed of the train. In case it exceeds the permitted vehicle speed (or a fault within the system is detected), the emergency braking is activated to stop the train. Other equipments, such as the Juridical Recording Unit (JRU) and odometry units, which measure and estimate the

speed and position of the vehicle, are also used on-board. Usually, a HMI is provided to the driver, denoted Man Machine Interface (MMI). These elements are connected to the EVC.

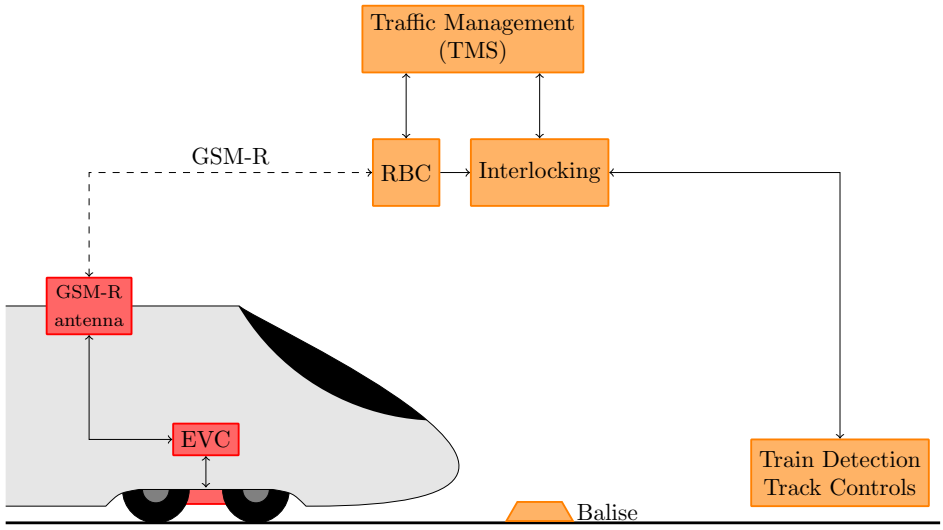


Figure 5.9: High-level ERTMS railway signalling system for levels 2 and 3

The radio messages sent from the train are collected by the Radio Block Center (RBC). This information is then forwarded to the Traffic Management System (TMS). The TMS manages the interlocking systems, train detection and track controls, as well as balises, installed on the track-side. It is also able to deliver safety-related information, such as movement authorities, to the vehicles through the GSM-R communications.

For the compatibility of the new system with former national railway management systems, two more levels are established. On the one hand, in ERTMS level 0 missions, ETCS equipped trains are operated along railway lines in which neither ERTMS nor a national signalling system is present. On the other hand, the ERTMS level National Train Control (NTC) refers to trains equipped with the ETCS system which are run in lines equipped with a national signalling system.

From the security point of view, the authentication and integrity of such radio messages transmitted among trains and track-side equipments have to be ensured. However, as affirmed by *I. Lopez* and *M. Aguado* [150], presently adopted security measures do not provide enough trustworthiness. Consequently, an update of such security mechanisms is necessary.

In this case study, an overview of GSM-R communications is firstly given and the security measures adopted to be adopted in these communications

then explained. The need of security measure replacements is also motivated. Specifically, a new cryptographic algorithm as an alternative to the one used in the Euroradio protocol is proposed. After that, a live patching process to replace such cryptographic algorithm is presented. This update is dynamically carried out by means of *Cetratus*.

5.3.2 GSM-R communications

GSM-R technology, which is an adaptation of the Global System for Mobile Communications (GSM) for the railway domain, enables the information exchange among trains and track-side systems in ERTMS level two and above. In order to prevent interferences with other radio communications, an specific frequency range is used by GSM-R, where the A5/1 encryption algorithm is applied (the A5/3 block cipher may also be supported). This cryptographic mechanism provides data confidentiality among mobile and base stations. Two layers are adopted on top of GSM-R in ERTMS: **Euroradio** and **Application Layer**. The communication layers used in ERTMS are illustrated in Figure 5.10.

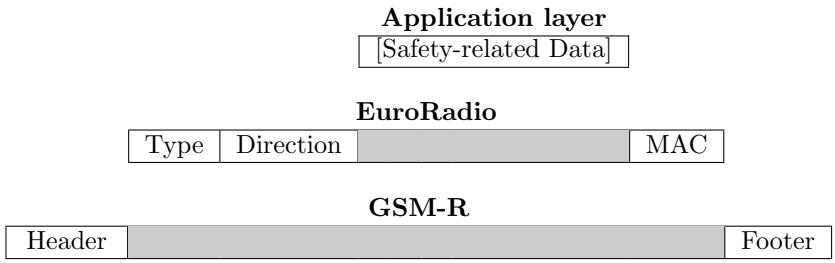


Figure 5.10: Communication layers in ERTMS

At the application layer, the safety-related messages are generated for data exchange between the trackside and the train. This application protocol provides protection against message delays, replay, malformation or deletion. To achieve this aim, message acknowledgements and timestamps are used. The specifications of this layer are given by the *Subset-026* [151] technical document. On the contrary, a MAC is provided by the Euroradio layer to ensure the authentication and integrity of the messages. For this end, the Triple Data Encryption Algorithm (TDEA) block cipher, also known as 3DES, is used on a CBC-MAC mode (CMAC). In TDEA, the Data Encryption Standard (DES) encryption algorithm is applied three times to each data block. As a result, a 64 bit MAC is generated. This procedure is

specified at the *subset-037* [152] technical document, where a 192 bit key, named *KsMAC* is necessary.

The selection of a more cost-efficient and secure cryptographic algorithm instead of TDEA is recommended by ENISA [153] and NIST [154]. As an alternative, the AES is proposed by the NIST. Furthermore, a key shorter than 128 bits will be not recommended by security agencies after 2020 [150]. A proof-of-concept attack against the Euroradio layer was presented by *Chothia et al.* [155], showing the security weaknesses of the actual ERTMS communications. In this work, a Movement Authority (MA) command was forged. In a similar way, the use of a different cryptographic primitive update, such as AES, is proposed to enhance the security level of the Euroradio layer. A 128 bits MAC is produced by the AES-128-CBC-MAC scheme with a 128-bit length key.

5.3.3 Live MAC Algorithm Update

In this section, a live patch is applied to the system through *Cetratus*, in which the GSM-R communication stack component is updated. The goal of this procedure is to bring up to date the former Euroradio TDEA based MAC algorithm with the recommended AES-128-CBC-MAC one. This case study was performed on a testing workbench, within a laboratory. A 32 bits single-core PowerPC embedded platform is employed and the *INTEGRITY* RTOS is used as the underlying operating system. The developed prototype is connected to a testing platform, from which inputs and outputs are driven. GSM-R communications are emulated as serial communications, since the developed prototype does not incorporate any GSM-R transceiver. In the final implementation, a GSM-R hardware module would be coupled to this serial interface.

In this live update process, a dynamic patch is firstly created and transferred then to the target system. In this dynamic patch, a new Euroradio component is enclosed. For compatibility reasons, the new application module processes both Euroradio MAC and AES-128-CBC-MAC schemes based messages. As depicted in Figure 5.11, four MA commands are sent to the train. The acquired authorisation speed is also visualized. In this case study, the Euroradio application component is cyclically executed every 30 ms.

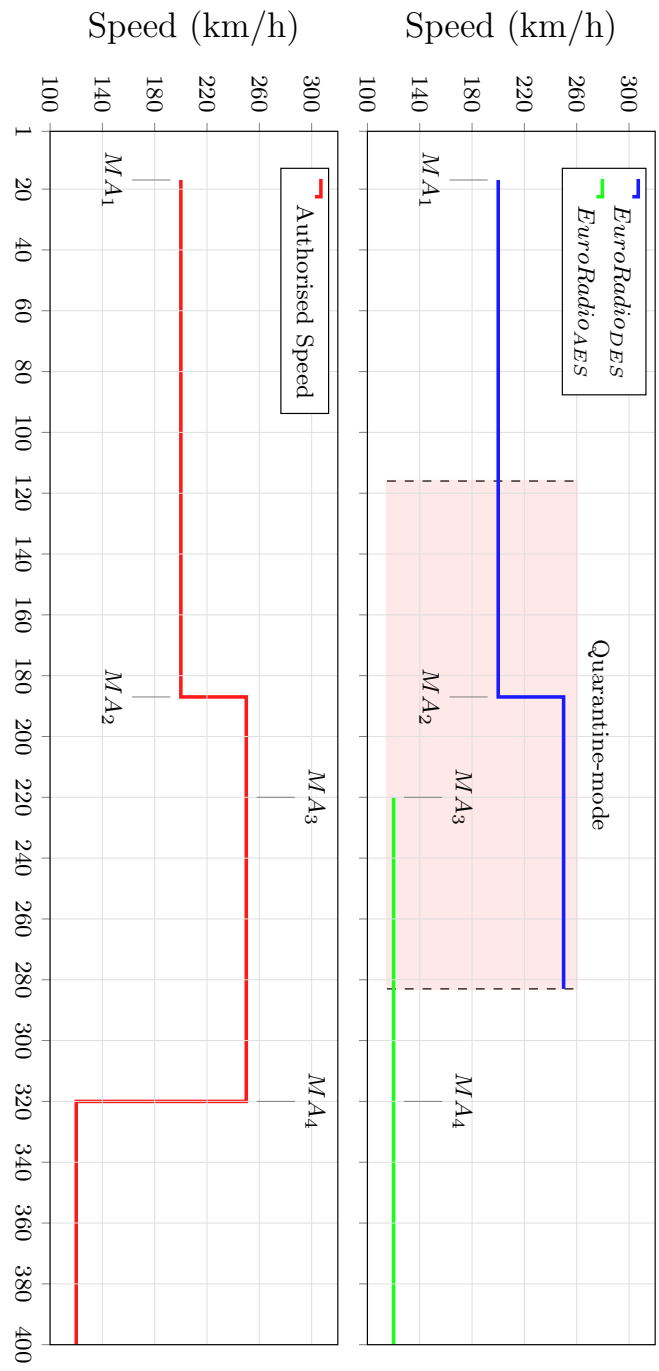


Figure 5.11: Computed and authorised train speeds

The first half of the transmitted MA messages include a 64 bit MAC, computed by applying the TDEA-based Euroradio MAC algorithm. In contrast, the remaining messages incorporated a 128 bit MAC, calculated using the AES-128-CBC-MAC scheme. These messages are given in Table 5.2. Meanwhile, eleven Position Report (PR) messages are transmitted from the train to the base station. These messages are not depicted in Figure 5.11.

Packet ID	**Safety-related Data**			**MAC algorithm**
	T_TRAIN	V_STATIC	V_LOA	
MA_1	1153	200 km/h	200 km/h	Euroradio MAC
MA_2	1159	250 km/h	250 km/h	
MA_3	1161	120 km/h	100 km/h	AES-128-CBC
MA_4	1165	120 km/h	100 km/h	

Table 5.2: Received Movement Authority (MA) messages by the train from the base station

As it can be observed in Figure 5.11, the Euroradio component firstly obtains and processes the MA_1 message. The authorised speed for the train is then settled to 200 km/h. After that, as request by the *Updater*, the new Euroradio component is initialized at cycle 115 and the quarantine-mode execution and monitoring starts. During this mode, two commands are received by the train: MA_2 and MA_3. On the one hand, MA_2 is processed by both $Euroradio_{DES}$ and $Euroradio_{AES}$ component Nevertheless, because a MAC based on the former Euroradio MAC scheme is employed, the safety data contained in the message is discarded by the $Euroradio_{AES}$ component. A maximum speed permission of 250 km/h is then accepted. On the other hand, the MA_3 message can not be processed by the old $Euroradio_{DES}$ component, since it can not corroborate the authenticity of it. The MA_3 message is not then taken into account by the system, since during the quarantine-mode period, the information processed by the new software $Euroradio_{AES}$ is not considered. The safety-related command is then omitted by the system. Therefore, in order to address this problem, two MA_3 messages containing the same safety-related information, each of them authenticated with a different MAC algorithm, should be transmitted by the track-side equipment to the train. As illustrated in Figure 5.11, while quarantine-mode execution, a limited speed of 120 km/h is obtained by the $Euroradio_{AES}$. Finally, once the behaviour of $Euroradio_{AES}$ is validated by the *Auditor*, the substitution is performed and the new $Euroradio_{AES}$

component is executed as the primary one. This is carried out at cycle 283. Afterwards, when the MA_4 is received, which contains the same safety-related data of MA_3, the authorised speed is set to 120 km/h.

Table 5.3 shows the transmitted PRs from the train to the base station, which contains the actual position and speed information of the vehicle.

Packet ID	**Safety-related Data**		**MAC algorithm**
	T_TRAIN	**V_TRAIN**	
PR_1	1154	100 km/h	Euroradio-MAC
PR_2	1155	100 km/h	
PR_3	1156	101 km/h	
PR_4	1157	101 km/h	
PR_5	1158	102 km/h	
PR_6	1160	102 km/h	
PR_7	1162	103 km/h	
PR_8	1163	103 km/h	
PR_9	1164	104 km/h	AES-128-CBC-MAC
PR_{10}	1166	103 km/h	
PR_{11}	1167	103 km/h	

Table 5.3: Transmitted PR messages from the train to the base station

Figure 5.12 shows the CPU time employed in each execution cycle for the old $Euroradio_{DES}$ and new $Euroradio_{AES}$ components. The CPU time used to receive and process MA messages, as well as to construct and send PR messages is displayed. Besides, the CPU time used by the *Cetratus* runtime is also presented. By default, a small overhead is introduced by *Cetratus* to the system. Yet, a significant increment is detected at cycle 115, where the setup of the patch is accomplished, up 13,56 % of CPU usage. As far the Euroradio component is concerned, as confirmed by *Chothia et al.* [155], the AES-128-CBC-MAC algorithm is executed much faster than the former EuraRadio MAC scheme. In the worst case, a CPU usage of 13,02 % is reached at cycle 136 while the quarantine-mode execution and monitoring, specifically, when the PR_4 message is assembled and transmitted.

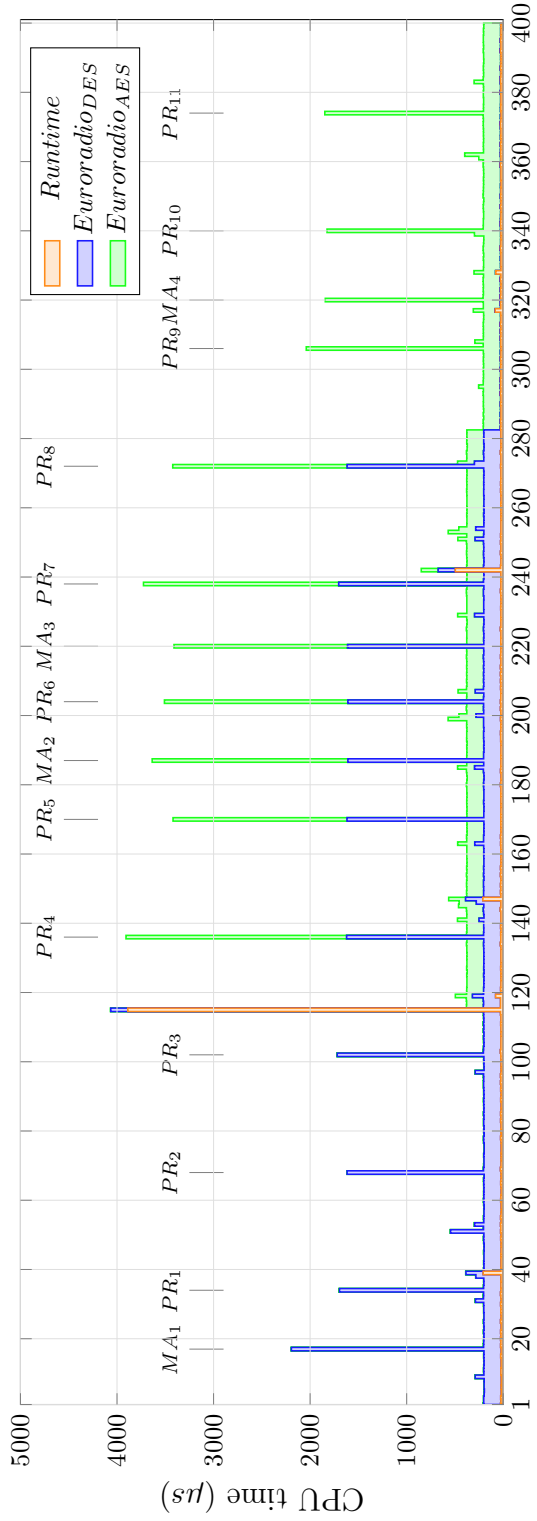


Figure 5.12: GSM-R message processing CPU time (*INTEGRITY* RTOS, executed on a 32 bits single-core PowerPC embedded platform)

6 Conclusions

High interconnectivity is desired in the actual IIoT era, also known as Industry 4.0 or the fourth industrial revolution. In this automation trend, which is referred as the industrial subset of the IoT, machine learning and big data technologies are incorporated to the already existing automation technologies, where connectivity, data acquisition and machine to machine communications are essential. However, due to the high interconnectivity among all connected devices, security concerns arise, specially for safety-critical systems, which deal with those scenarios that might lead to serious injury to people, loss of life, or damage to the natural environment. Consequently, the safety engineering community has started to take into account those security threats. It has to be ensured that these mixed-criticality systems provide not only the required safety integrity level, but also be resilient against cyber-attacks.

Safety-critical systems have usually long operational periods, up to twenty or thirty years from time to time, and might require a recertification process after any modification. Due to these reasons, software updates are rarely applied to these systems. However, even that leading edge security countermeasures are adopted at the development phase, these protection mechanisms could be overtaken sometime, since new security vulnerabilities are discovered every day. Consequently, software updates are necessary. Nevertheless, system shutdown and restarts may not be acceptable in those safety-related applications where high-availability is requested.

6.1 Contribution

This PhD thesis provides a review of industrial safety and security standards regarding software updates. Besides, a literature review on DSU techniques and systems for safe and secure IACS is given. In this study, twenty different DSU systems were analysed, categorised and compared. The DSU properties of each DSU system was also examined and the compliance against safety and security requirements evaluated.

Besides, a novel software framework, called *Cetratus*, is proposed for safety-critical systems in this PhD thesis, which enables the dynamic update of application components. The fundamental characteristic is the quarantine-mode, where the initialization and the execution of the soft-

ware patch is isolated. As a result, patching failures do not lead to any unsafe scenario or disruption of the service. The proposed solution is able to enhance current security protection measures without system shutdown. *Cetratus* is aligned with the industrial IEC 61508 [13] and IEC 62443 [14] standards and satisfies the internal test activity recommended by the IEC 62443-2-3: Patch management in the IACS environment technical document [15]. As required by this standard, the integrity and authenticity of the dynamic patch is checked before proceeding with the update. Table 6.1 shows the dynamic software updating methods utilized in *Cetratus*.

Characteristic	**Property**	**Employed Method**
Code transformation	Technique	Indirection handling
	Unit of update	Component
State transformation	State transformer	Manual
	Mode	Eagerly
	Data update	Checkpointing
Update point	Specification	Activeness Safety
	Multi-threaded	No

Table 6.1: Dynamic software updating methods employed in *Cetratus*

A proof-of-concept of *Cetratus* has been implemented as an Ada runtime service. The Ada programming language is highly recommended by the safety and even security engineering communities for the development of safety-critical and/or mission-critical systems. High portability among operating systems, and thus hardware platforms, is achieved firstly by means of the selected programming language, and secondly, by virtue of the POSIX API. *Cetratus* prototype has been successfully built for x86 and PowerPC architectures. The initial validation of the prototype was performed on a x86 computer, running the Real-Time Linux operating system, where the signal processing algorithm was dynamically modified. The prototype was then integrated with Integrity RTOS and executed on a x86 industrial computer and on a 32 bits single-core PowerPC embedded platform.

Furthermore, two case studies are presented. On the one hand, a mixed-criticality solution based on the *Cetratus* is presented for a BEMS. This system observes and manages different energy sources and outcomes on a residential building, where energy-related data, such as energy savings and consumption measurements, are transmitted to a BEOS cloud application. In the presented live update example, an homomorphic encryption algorithm is included in the new application component version. The experiment is performed on the x86 industrial computer.

On the other hand, in the railway case study, the former MAC scheme of the Euroradio application component, for which security flaws have been disclosed, is upgraded. As an alternative, a MAC algorithm based the AES block cipher, a cryptographic algorithm suggested by NIST, is employed. This new MAC scheme would allow to ensure the integrity and authenticity of the GSM-R messages exchanged among trains and track-side ERTMS systems. The live update is carried out on the 32 bits single-core PowerPC embedded platform, on a testing workbench.

This research topic has garnered attention in recent years during the preparation of this PhD thesis. A patent was fulfilled and granted by/to the *National Technology & Engineering Solutions of Sandia, LLC* [156], where a system and a method for real-time upgrade of industrial control software is presented. The same approach and DSU techniques employed in/for *Cetratus* are used. Nevertheless, it does not provide software patch monitoring features and the whole program is updated at once. A re-certification of the system would be then needed. A multi-version execution approach based DSU system, called *MVEDSUA*, was also proposed by *L. Pina et al.* [157] (in collaboration with *A. Andronidis*, *M. Hicks*, the author of DLpop, and *C. Cadar* [79, 80]), which targets high-performance servers. To this end, they extended the Kitsune DSU system to enable multi-version execution. A leader-follower strategy is adopted, which is analogous to the quarantine-mode based approach presented in this PhD thesis. Figure 6.1 shows the DSU stages in *MVEDSUA* [157].

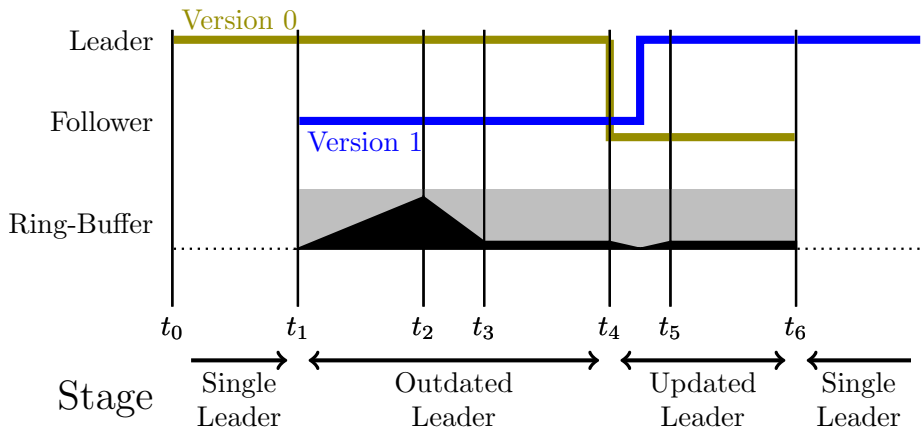


Figure 6.1: Update stages of *MVEDSUA* (reproduced from ref. [157] with permission from *L. Pina*)

6.2 Dissemination

In this section, the peer-reviewed conference papers and journal articles published during this thesis are shown. Actually, some passages in this PhD thesis have been quoted verbatim from these sources.

Conference Papers

Title: Software Updates in Safety and Security Co-engineering

Authors: Imanol Mugarza, Jorge Parra and Eduardo Jacob

Conference: SAFECOMP2017: International Conference on Computer Safety, Reliability, and Security. 12^{th} International ERCIM/EWICS/ARTEMIS Workshop on "Dependable Smart Embedded and Cyber-physical Systems and Systems-of-Systems".

Publication: Lecture Notes in Computer Science book seriec (LNCS, volume 10489, pp. 199-210)

Date & Venue: September 12, 2017 - Trento (Italy)

Title: Cetratus: Towards a live patching supported runtime for mixed-criticality safe and secure systems

Authors: Imanol Mugarza, Jorge Parra and Eduardo Jacob

Conference: SIES 2018: International Symposium on Industrial Embedded Systems

Publication: IEEE proceedings (pp. 1-8)

Date & Venue: June 6-8, 2018 - Graz (Austria)

Journal Articles

Title: Analysis Of Existing Dynamic Software Updating Techniques For Safe And Secure Industrial Control Systems

Authors: Imanol Mugarza, Jorge Parra and Eduardo Jacob

Journal: International Journal of Safety and Security Engineering (ISSN: 2041-904X)

Quality: SJR in 2018: 0.163, Q3

DOI: `https://doi.org/10.2495/SAFE-V8-N1-121-131`

Editorial: WIT Press

Date: February 1, 2018

Title: Dynamic Software Updates to Enhance Security and Privacy in High Availability Energy Management Applications in Smart Cities

Authors: Imanol Mugarza, Andoni Amurrio, Ekain Azketa and Eduardo Jacob

Journal: IEEE Access (ISSN: 2169-3536)

Quality: JCR in 2018: 4.098, Q1

DOI: `https://doi.org/10.1109/ACCESS.2019.2905925`

Editorial: IEEE

Date: March 21, 2019

Journal Articles under Review

Title: Cetratus: A framework for zero downtime secure software updates in safety-critical systems

Authors: Imanol Mugarza, Jorge Parra and Eduardo Jacob

Journal: Software: Practice and Experience

Quality: SJR in 2018: 0.45, Q2

Editorial: Wiley

6.3 Future Work

Although a dynamic software update mechanism for zero downtime safety-critical systems is proposed in this thesis, further research is necessary in order to be able to accomplish live software updates on computer-based safety-critical systems.

On the hand one, procedures and methods to verify and validate the new software versions are necessary. The validation of a new software version is a challenging task, which highly depends on the deployed application. It has to be ensured that the new software components fulfil with the expected functionality and requirements. To this end, a regression testing technique, which is defined by *G. Rothermel* [158, 159] as "an expensive task performed on modified software to provide confidence that modified code behaves as intended, and that modifications have not inadvertently disrupted the of unmodified code", might be used. Usually, in regression testing, previously generated test-suits are used instead of generating new ones. With this in mind, the *Auditor* should gather software updates monitoring data and evaluate, for example by means of statistical analyses or formal methods, as proposed by *M. Jalili* [160], the correctness of the new software version. A general testing framework was proposed by *L. Pina* and *M. Hicks* [161] to systematically test if the dynamic update introduces any undesired misbehaviours and/or crashes. Besides, the specifications of a secure remote dynamic software updating service could be defined.

On the other hand, as reported by *J.L. Fenn et al.* [162], actual re-certification costs depend on the size and complexity of the system to be updated. Consequently, even that a minimal change is performed, re-certification costs might reach or exceed the initial assessment costs. As stated, the goal is to relate those costs to the characteristics of the change itself, instead to the system. Based on the feedback and lessons learned [162], in order to reduce re-certification costs and time to market, previously generated engineering artefacts might be reused.

As indicated by *J.L. Fenn et al.* [162], a successful elaboration of a modular assurance case is firstly mandatory in order to achieve an incremental certification. This property is also a key factor for the maintainability of the assurance cases. Modularity enables broad granularity and high cohesion among system components. The use of COTS products is also propitious. The design of the assurance case starts at the early system development phases. In this step, a study focusing on the expected or possible changes to the system is crucial, for example due to the new functional security requirements. This analysis allows to build assurance cases which are more receptive to changes. In case no change scenario is contemplated at all, it is conceived that the system will remain unchangeable over its lifetime.

Figure 6.2 illustrates the conceptual idea of a modular and incremental certification approach. The fundamental idea is to reuse engineering artefacts from a previously accomplished certification process. The volume of the required new artefacts subset might also be tried to reduce. This procedure would repeatedly be achieved for each system software upgrade. Notice that performing a software upgrade is a risk and business management decision [162].

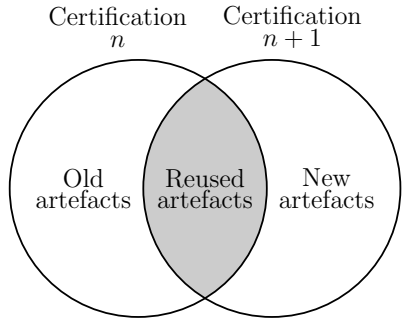


Figure 6.2: Reuse of engineering artefacts in an incremental certification

After categorizing and filtering each software modification, system maintainers shall determine which changes are required for the system. For the re-certification of the system, it is crucial to came upon with highly maintainable and long living assurance cases, where new argumentation elements and therefore new evidences, are incorporated. As stated by *Larrucea* [71], the modularity strategy permits the reuse of safety case elements and components. An additive argumentation approach is necessary. Moreover, historical evidences, argumentation and certification data for each system certification shall also be preserved. In the context of this thesis, these evidence gathering and argumentation tasks should be carried out by the *Auditor* user.

Bibliography

[1] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, vol. 42, no. 4, pp. 0042–52, 2009.

[2] A. Sangiovanni-Vincentelli and M. Di Natale, "Embedded system design for automotive applications," *IEEE Computer*, vol. 40, no. 10, pp. 42–51, 2007.

[3] P. Feiler, J. Goodenough, A. Gurfinkel, C. Weinstock, and L. Wrage, "Four pillars for improving the quality of safety-critical software-reliant systems," tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2013.

[4] S. Paul and L. Rioux, "Over 20 years of research in cybersecurity and safety engineering: a short bibliography,"

[5] S. Paul, "On the meaning of security for safety (s4s)," *WIT Transactions on The Built Environment*, vol. 151, pp. 379–389, 2015.

[6] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security Privacy*, vol. 9, pp. 49–51, May 2011.

[7] L. Piètre-Cambacédès and C. Chaudet, "The sema referential framework: Avoiding ambiguities in the terms "security" and "safety"," *International Journal of Critical Infrastructure Protection*, vol. 3, no. 2, pp. 55–66, 2010.

[8] INTERNATIONAL ATOMIC ENERGY AGENCY, *IAEA Safety Glossary*. Vienna: INTERNATIONAL ATOMIC ENERGY AGENCY, 2008.

[9] N. Kuntze, C. Rudolph, G. B. Brisbois, M. Boggess, B. Endicott-Popovsky, and S. Leivesley, "Security vs. safety: Why do people die despite good safety?," in *Integrated Communication, Navigation, and Surveillance Conference (ICNS), 2015*, pp. A4–1, IEEE, 2015.

[10] P.-c. Ludovic and C. Claude, "Disentangling the relations between safety and security,"

[11] A. Burns, J. McDermid, and J. Dobson, "On the meaning of safety and security," *The Computer Journal*, vol. 35, no. 1, pp. 3–15, 1992.

[12] Kaspersky Security Intelligence, "Industrial cybersecurity threat landscape," 2016. [Online; Accessed 19. November 2016].

[13] International Electrotechnical Commission, "Functional safety of electrical/electronic/programmable electronic safety related systems," *IEC 61508*, 2000.

[14] International Electrotechnical Commission, "IEC 62443: Industrial communication networks - Network and system security," 2010.

[15] International Electrotechnical Commission, "IEC 62443-2-3: Industrial communication networks - Network and system security - Patch management in the IACS environment," 2010.

[16] É. Leverett, R. Clayton, and R. Anderson, "Standardisation and certification of the 'internet of things'," 2017.

[17] International Electrotechnical Commission and others, "IEC 61784-3: Functional safety fieldbuses," *ed. GENEVA, SWITZERLAND: IEC Central Office*, 2010.

[18] Y. Beres and J. Griffin, "Optimizing network patching policy decisions," *Information Security and Privacy Research*, pp. 424–442, 2012.

[19] A. Pauna and K. Moulinos, "Windows of exposure... a real problem for scada systems," tech. rep., Technical report, ENISA (Dec. 2013), 2013.

[20] R. Leszczyna, E. Egozcue, L. Tarrafeta, V. F. Villar, R. Estremera, and J. Alonso, "Protecting industrial control systems-recommendations for europe and member states," tech. rep., Technical report, European Union Agency for Network and Information Security (ENISA), 2011.

[21] S. Tom, D. Christiansen, and D. Berrett, "Recommended practice for patch management of control systems," *DHS control system security program (CSSP) Recommended Practice*, 2008.

[22] J. Thomson, *High Integrity Systems and Safety Management in Hazardous Industries*. Butterworth-Heinemann, 2015.

[23] B. Kesler, "The vulnerability of nuclear facilities to cyber attack," *Strategic Insights*, vol. 10, no. 1, pp. 15–25, 2011.

[24] M. Holt, R. J. Campbell, and M. B. Nikitin, *Fukushima nuclear disaster.* Congressional Research Service, 2012.

[25] W. J. Garland, "Decay heat estimates for mnr," *McMaster Nuclear Reactor, McMaster University, Ontario*, 1999.

[26] H. Khurana, M. Hadley, N. Lu, and D. A. Frincke, "Smart-grid security issues," *IEEE Security & Privacy*, vol. 8, no. 1, 2010.

[27] R. Anderson, *Security engineering.* John Wiley & Sons, 2008.

[28] "Itu-tx.1205: series x: data networks, open system communications and security: telecommunication security: overview of cybersecurity.," tech. rep., International Telecommunications Union (ITU), 2008.

[29] K. Stouffer, J. Falco, and K. Scarfone, "Guide to industrial control systems (ics) security," *NIST special publication*, vol. 800, no. 82, pp. 16–16, 2011.

[30] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography.* CRC press, 1996.

[31] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C.* john wiley & sons, 2007.

[32] L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, *Report on post-quantum cryptography.* US Department of Commerce, National Institute of Standards and Technology, 2016.

[33] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. Dray Jr, "Advanced encryption standard (aes)," tech. rep., 2001.

[34] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library-seal v2. 1," in *International Conference on Financial Cryptography and Data Security*, pp. 3–18, Springer, 2017.

[35] S. Zhangy, A. Kim, D. Liu, S. C. Nuckchadyy, L. Huangy, A. Masurkary, J. Zhangy, L. P. Karnatiz, L. Martinezx, T. Hardjono, *et al.*, "Genie: A secure, transparent sharing and services platform for genetic and health data," *arXiv preprint arXiv:1811.01431*, 2018.

[36] W. E. Burr, D. F. Dodson, E. M. Newton, R. A. Perlner, W. T. Polk, S. Gupta, and E. A. Nabbus, "Sp 800-63-1. electronic authentication guideline," 2011.

[37] E. Yuan and J. Tong, "Attributed based access control (abac) for web services," in *IEEE International Conference on Web Services (ICWS'05)*, IEEE, 2005.

[38] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The nist model for role-based access control: towards a unified standard," in *ACM workshop on Role-based access control*, vol. 2000, 2000.

[39] S. Suehring, *Linux Firewalls: Enhancing Security with Nftables and Beyond*. Addison-Wesley, 2015.

[40] L. Gheorghe, *Designing and Implementing Linux Firewalls with QoS using netfilter, iproute2, NAT and l7-filter*. Packt Publishing Ltd, 2006.

[41] K. Scarfone and P. Mell, "Guide to intrusion detection and prevention systems (idps)," tech. rep., National Institute of Standards and Technology, 2012.

[42] A. Syalim, Y. Hori, and K. Sakurai, "Comparison of risk analysis methods: Mehari, magerit, nist800-30 and microsoft's security management guide," in *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, pp. 726–731, IEEE, 2009.

[43] M. Souppaya and K. Scarfone, "Guide to enterprise patch management technologies," *NIST Special Publication*, vol. 800, p. 40, 2013.

[44] D. J. Smith and K. G. Simpson, *Safety Critical Systems Handbook: A Straightfoward Guide To Functional Safety, Iec 61508 (2010 Edition) And Related Standards, Including Process Iec 61511 And Machinery Iec 62061 And Iso 13849*. Elsevier, 2010.

[45] B. Alessandro, "Reliability engineering theory and practice," 2010.

[46] M. Rausand, H. Arnljot, *et al.*, *System reliability theory: models, statistical methods, and applications*, vol. 396. John Wiley & Sons, 2004.

[47] M. L. Shooman, *Reliability of Computer Systems and Networks: Fault Tolerance,Analysis,and Design*. New York, NY, USA: John Wiley & Sons, Inc., 2002.

[48] J. Rushby, "Formally verified hardware encapsulation mechanism for security, integrity, and safety," tech. rep., DTIC Document, 2002.

[49] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms, and assurance," tech. rep., DTIC Document, 2000.

[50] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," in *11th Real-Time Linux Workshop*, pp. 263–272, Citeseer, 2009.

[51] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The xtratum approach," in *Dependable Computing Conference (EDCC), 2010 European*, pp. 67–72, IEEE, 2010.

[52] E. Blasch, P. Kostek, P. Pačes, and K. Kramer, "Summary of avionics technologies," *IEEE Aerospace and Electronic Systems Magazine*, vol. 30, no. 9, pp. 6–11, 2015.

[53] A. Crespo, A. Alonso, M. Marcos, A. Juan, and P. Balbastre, "Mixed criticality in control systems," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 12261–12271, 2014.

[54] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, pp. 203–216, ACM, 1994.

[55] B. Hunter, "Integrating safety and security into the system lifecycle," in *Improving Systems and Software Engineering Conference (ISSEC), Canberr, Australia*, p. 147, 2009.

[56] S. Paul, L. Rioux, T. Wiander, and F. Vallée, "Recommendations for security and safety co-engineering (release n 2)," *ITEA2 MERgE project*, 2015.

[57] L. Piètre-Cambacédès and M. Bouissou, "Cross-fertilization between safety and security engineering," *Reliability Engineering & System Safety*, vol. 110, pp. 110–126, 2013.

[58] S. Kriaa, L. Pietre-Cambacedes, M. Bouissou, and Y. Halgand, "A survey of approaches combining safety and security for industrial control systems," *Reliability Engineering & System Safety*, vol. 139, pp. 156 – 178, 2015.

[59] B. Hunter, "Assuring separation of safety and non-safety related systems," in *Proceedings of the eleventh Australian workshop on Safety critical systems and software-Volume 69*, pp. 45–51, Australian Computer Society, Inc., 2007.

[60] H.-H. Bock, J. Braband, B. Milius, and H. Schäbe, *Towards an IT Security Protection Profile for Safety-Related Communication in Railway Automation*, pp. 137–148. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[61] J. Åkerberg, M. Gidlund, T. Lennvall, J. Neander, and M. Björkman, "Efficient integration of secure and safety critical industrial wireless sensor networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2011, p. 100, Sep 2011.

[62] F. Wieczorek, C. Krauß, F. Schiller, and C. Eckert, "Towards secure fieldbus communication," in *Computer Safety, Reliability, and Security* (F. Ortmeier and P. Daniel, eds.), (Berlin, Heidelberg), pp. 149–160, Springer Berlin Heidelberg, 2012.

[63] T. Kelly, "A systematic approach to safety case management," tech. rep., SAE Technical Paper, 2004.

[64] J. Goodenough, H. Lipson, and C. Weinstock, "Arguing security-creating security assurance cases," *rapport en ligne (initiative build security-in du US CERT), Université Carnegie Mellon*, 2007.

[65] P. J. Graydon and T. P. Kelly, "Using argumentation to evaluate software assurance standards," *Information and Software Technology*, vol. 55, no. 9, pp. 1551–1562, 2013.

[66] Y. He and C. Johnson, "Generic security cases for information system security in healthcare systems," 2012.

[67] C. Preschern, "Catalog of security tactics linked to common criteria requirements," in *Proceedings of the 19th Conference on Pattern Languages of Programs*, p. 7, The Hillside Group, 2012.

[68] T. P. Kelly, *Arguing safety: a systematic approach to managing safety cases*. PhD thesis, University of York, 1999.

[69] T. Kelly, I. Bate, J. McDermid, and A. Burns, "Building a preliminary safety case: An example from aerospace," *ROLLS ROYCE PLC-REPORT-PNR*, 1998.

[70] P. Graydon and I. Bate, "Realistic safety cases for the timing of systems," *The Computer Journal*, vol. 57, no. 5, pp. 759–774, 2013.

[71] A. Larrucea, I. Martinez, C. F. Nicolas, J. Perer, and R. Obermaisser, "Modular development and certification of dependable mixed-criticality systems," in *2017 Euromicro Conference on Digital System Design (DSD)*, pp. 419–426, IEEE, 2017.

[72] SANS Institute, "Common Criteria and Protection Profiles: How to Evaluate Information," 2003.

[73] M. Nicholson, P. Conmy, I. Bate, and J. McDermid, "Generating and maintaining a safety argument for integrated modular systems," in *5th Australian Workshop on Industrial Experience with Safety Critical Systems and Software, Melbourne, Australia*, pp. 31–41, 2000.

[74] SANS Institute, "The Common Criteria ISO/IEC 15408 - The Insight, Some Thoughts, Questions and Issues," 2001.

[75] M. Hicks, J. T. Moore, and S. Nettles, *Dynamic software updating*, vol. 36. ACM, 2001.

[76] K. Makris, *Whole-program dynamic software updating.* PhD thesis, Arizona State University, 2009.

[77] J. Mitchell, D. Henderson, and G. Ahrens, "Ibm power5 processor-based servers: A highly available design for business-critical applications," *IBM White paper*, 2005.

[78] D. Henderson, J. Mitchel, and G. Ahrens, "Power7 r system ras: Key aspects of power systemst m reliability, availability, and serviceability," 2010.

[79] C. Cadar and P. Hosek, "Multi-version software updates," in *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, pp. 36–40, IEEE Press, 2012.

[80] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 612–621, IEEE Press, 2013.

[81] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for c," in *ACM SIGPLAN Notices*, vol. 47, pp. 249–264, ACM, 2012.

[82] L. G. G. de Pina, *Practical Dynamic Software Updating.* PhD thesis, INSTITUTO SUPERIOR TECNICO, 2016.

[83] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.," in *Usenix Security*, vol. 98, pp. 63–78, 1998.

[84] S. Sinha, M. Koedam, R. Van Wijk, A. Nelson, A. B. Nejad, M. Geilen, and K. Goossens, "Composable and predictable dynamic loading for time-critical partitioned systems," in *2014 17th Euromicro Conference on Digital System Design*, pp. 285–292, IEEE, 2014.

[85] S. Subramanian, M. Hicks, and K. S. McKinley, *Dynamic software updates: a VM-centric approach*, vol. 44. ACM, 2009.

[86] M. Solarski, "Dynamic upgrade of distributed software components," 2004.

[87] Y. Berbers and Y. Vandewoude, "Dynamically updating component-oriented systems.," 2007.

[88] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp. 72–83, ACM, 2006.

[89] K. Saur, M. Hicks, and J. S. Foster, "C-strider: type-aware heap traversal for c," *Software: Practice and Experience*, 2015.

[90] K. Saur, "Dynamic upgrades for high availability systems," 2015.

[91] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, "State transfer for clear and efficient runtime updates," in *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pp. 179–184, IEEE, 2011.

[92] C. M. Hayden, "Clear, correct, and efficient dynamic software updates," 2012.

[93] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "Opus: Online patches and updates for security.," in *Usenix Security*, vol. 5, p. 18, 2005.

[94] G. Stoyle, "A theory of dynamic software updates," 2007.

[95] "ISO/DIS 26262 - Road vehicles – Functional safety," tech. rep., Geneva, Switzerland, July 2009.

[96] International Electrotechnical Commission, "IEC 62278: Railway Applications—Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS)," *ed. GENEVA, SWITZERLAND: IEC Central Office*, pp. 21–24, 2002.

[97] International Electrotechnical Commission, "IEC 62279: Railway applications-Software for railway control and protection systems," *ed. GENEVA, SWITZERLAND: IEC Central Office*, 2002.

[98] International Electrotechnical Commission, "IEC 62425: Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling," *ed. GENEVA, SWITZERLAND: IEC Central Office*, pp. 21–24, 2002.

[99] "IEC 61511 Functional Safety - Safety instrumented systems for the process industry sector," tech. rep., International Electrotechnical Commission, 2003.

[100] International Electrotechnical Commission, "IEC 62061: Safety of machinery—Functional safety of safetyrelated electrical, electronic and programmable electronic control systems," *IEC*, 2005.

[101] International Electrotechnical Commission and others, "IEC 61784: Digital data communications for measurement and control," *ed. GENEVA, SWITZERLAND: IEC Central Office*, pp. 21–24, 2010.

[102] G. Disterer, "ISO/IEC 27000, 27001 and 27002 for information security management," 2013.

[103] The Common Criteria Recognition Agreement Members, "Common Criteria for Information Technology Security Evaluation." http://www.commoncriteriaportal.org/, Sept. 2006.

[104] R. Melton, T. Fletcher, and M. Earley, "System protection profile–industrial control systems," *Version 1.0, National Institute of Standards and Technology*, 2004.

[105] Bundesamt für Sicherheit in der Informationstechnik, "Common criteria protection profile standard reader - smart card reader with pinpad supporting eid based on extended access control," *Bundesamt für Sicherheit in der Informationstechnik*, 2013.

[106] H. Seifzadeh, H. Abolhassani, and M. S. Moshkenani, "A survey of dynamic software updating," *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 535–568, 2013.

[107] E. Miedes and F. D. Munoz-Escoı, "Dynamic software update," tech. rep., Technical Report ITI-SIDI-2012/004, 2012.

[108] A. Richter, "Dlopen (3)," *Linux Programmer's Manual*, 1995.

[109] M. Payer, B. Bluntschli, and T. R. Gross, "Dynsec: On-the-fly code rewriting and repair.," in *HotSWUp*, 2013.

[110] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *Proceedings of the 29th international conference on Software Engineering*, pp. 271–281, IEEE Computer Society, 2007.

[111] K. Makris and R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction.," in *USENIX Annual Technical Conference*, vol. 2009, 2009.

[112] K. Makris, "Upstare manual," 2012.

[113] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *International Conference on Compiler Construction*, pp. 213–228, Springer, 2002.

[114] I. G. Neamtiu, *Practical Dynamic Software Updating*. ProQuest, 2008.

[115] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Live updating operating systems using virtualization," in *Proceedings of the 2nd international conference on Virtual execution environments*, pp. 35–44, ACM, 2006.

[116] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of nonquiescent subsystems in commodity operating system kernels," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 327–340, 2007.

[117] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th ACM European conference on Computer systems*, pp. 187–198, ACM, 2009.

[118] C. Binnie, "Zero downtime linux," in *Practical Linux Topics*, pp. 33–39, Springer, 2016.

[119] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, *et al.*, "K42: building a complete operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 133–145, 2006.

[120] A. Baumann, J. Appavoo, D. Da Silva, O. Krieger, and R. W. Wisniewski, "Improving operating system availability with dynamic update," in *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, pp. 21–27, 2004.

[121] A. Baumann, "Dynamic update for operating systems," *Doctor of Philosophy, School of Computer Science and Engineering, The University of New South Wales*, vol. 112, 2007.

[122] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr, "Providing dynamic update in an operating system.," in *USENIX Annual Technical Conference, General Track*, pp. 279–291, 2005.

[123] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 279–292, ACM, 2013.

[124] C. Giuffrida *et al.*, *Safe and Automatic Live Update*. VU University Amsterdam, 2014.

[125] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005.

[126] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[127] C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A. S. Tanenbaum, "Back to the future: Fault-tolerant live update with time-traveling state transfer.," in *LISA*, pp. 89–104, 2013.

[128] J. Montgomery, "A model for updating real-time applications," *Real-Time Systems*, vol. 27, no. 2, pp. 169–189, 2004.

[129] A. C. Noubissi, J. Iguchi-Cartigny, and J.-L. Lanet, "Hot updates for java based smart cards," in *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pp. 168–173, IEEE, 2011.

[130] G. Gracioli and A. A. Fröhlich, "An operating system infrastructure for remote code update in deeply embedded systems," in *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, p. 3, ACM, 2008.

[131] S. Kang, I. Chun, and W. Kim, "Dynamic software updating for cyber-physical systems," in *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*, pp. 1–3, June 2014.

[132] H. Seifzadeh, A. A. P. Kazem, M. Kargahi, and A. Movaghar, "A method for dynamic software updating in real-time systems," in *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*, pp. 34–38, June 2009.

[133] M. Wahler, S. Richter, and M. Oriol, "Dynamic software updates for real-time systems," in *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, p. 2, ACM, 2009.

[134] M. Wahler, S. Richter, S. Kumar, and M. Oriol, "Non-disruptive large-scale component updates for real-time controllers," in *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on Data Engineering Workshops*, pp. 174–178, IEEE, 2011.

[135] M. Oriol, M. Wahler, R. Steiger, S. Stoeter, E. Vardar, H. Koziolek, and A. Kumar, "Fasa: a scalable software framework for distributed control systems," in *Proceedings of the 3rd international ACM SIGSOFT symposium on Architecting Critical Systems*, pp. 51–60, ACM, 2012.

[136] M. Wahler, M. Oriol, and A. Monot, "Real-time multi-core components for cyber-physical systems," in *2015 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*, pp. 37–42, May 2015.

[137] A. Monot, M. Oriol, C. Schneider, and M. Wahler, "Modern software architecture for embedded real-time devices: High value, little overhead," in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 201–210, April 2016.

[138] M. Wahler, T. Gamer, A. Kumar, and M. Oriol, "Fasa: A software architecture and runtime framework for flexible distributed automation systems," *Journal of Systems Architecture*, vol. 61, no. 2, pp. 82–111, 2015.

[139] M. Wahler and M. Oriol, "Disruption-free software updates in automation systems," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pp. 1–8, IEEE, 2014.

[140] R. Pierce, *Preliminary Assessment of Linux for safety related systems*. HSE Books, 2002.

[141] N. Mc Guire, "Linux for safety critical systems in iec 61508 context," in *Proceedings of the Ninth Real-Time Linux Workshop in Linz*, 2007.

[142] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. v. Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," 2015.

[143] L. Baresi, C. Ghezzi, X. Ma, and V. P. La Manna, "Efficient dynamic updates of distributed components through version consistency," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 340–358, 2017.

[144] J. Garrido, J. Zamorano, and J. A. de la Puente, "Arinc-653 interpartition communications and the ravenscar profile," *ACM SIGAda Ada Letters*, vol. 35, no. 1, pp. 38–45, 2015.

[145] J. A. de la Puente, J. F. Ruiz, and J. M. González-Barahona, "Real-time programming with gnat: specialised kernels versus posix threads," in *ACM SIGAda Ada Letters*, vol. 19, pp. 73–77, ACM, 1999.

[146] H. Lund, P. A. Østergaard, D. Connolly, and B. V. Mathiesen, "Smart energy and smart energy systems," *Energy*, vol. 137, pp. 556–565, 2017.

[147] D. CAR, "Replicable and innovative future efficient districts and cities," *ENERGY*, vol. 2013, pp. 8–1.

[148] J. C. Palencia, M. G. Harbour, J. J. Gutiérrez, and J. M. Rivas, "Response-time analysis in hierarchically-scheduled time-partitioned distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, 2017.

[149] K. Zhang, J. Ni, K. Yang, X. Liang, J. Ren, and X. S. Shen, "Security and privacy in smart city applications: Challenges and solutions," *IEEE Communications Magazine*, vol. 55, no. 1, pp. 122–129, 2017.

[150] I. Lopez and M. Aguado, "Cyber security analysis of the european train control system," *IEEE Communications Magazine*, vol. 53, no. 10, pp. 110–116, 2015.

[151] UNISIG, "Subset- 026, system requirements specification, version 3.0.0."

[152] UNISIG, "Subset- 037, euroradio fis, version 3.2.0."

[153] N. P. Smart, V. Rijmen, B. Gierlichs, K. Paterson, M. Stam, B. Warinschi, and G. Watson, "Algorithms, key size and parameters report," *European Union Agency for Network and Information Security*, pp. 0–95, 2014.

[154] National Institure of Standards and Technology (NIST), "Itl bulletin for novenber 2017 - guidance on tdea block ciphers,"

[155] T. Chothia, M. Ordean, J. de Ruiter, and R. J. Thomas, "An attack against message authentication in the ertms train to trackside communication protocols," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 743–756, ACM, 2017.

[156] A. R. Chavez, K. Phan, J. Hosic, R. M. Birmingham, and J. D. Patel, "Real-time software upgrade," July 31 2018. US Patent App. 10/037,203.

[157] L. Pina, A. Andronidis, M. Hicks, and C. Cadar, "Mvedsua: Higher availability dynamic software updates via multi-version execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[158] G. Rothermel and M. J. Harrold, "Selecting tests and identifying test coverage requirements for modified software," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 169–184, ACM, 1994.

[159] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[160] M. Jalili Kordkandi, "Towards change validation in dynamic system updating frameworks," 2018.

[161] L. Pina and M. Hicks, "Tedsuto: A general framework for testing dynamic software updates," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 278–287, IEEE, 2016.

[162] J. L. Fenn, R. Hawkins, P. Williams, T. Kelly, M. Banner, and Y. Oakshott, "The who, where, how, why and when of modular and incremental certification," in *2nd IET International Conference on System Safety*, pp. 135–140, IET, 2007.