IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 25, 2019, accepted March 15, 2019, date of publication March 21, 2019, date of current version April 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2905925

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

IMANOL MUGARZA[1], ANDONI AMURRIO[1], EKAIN AZKETA[1], AND EDUARDO JACOB[2]
[1]Dependable Embedded Systems Area, IK4-IKERLAN Technology Research Centre, 20500 Arrasate-Mondragón, Spain
[2]Faculty of Engineering, University of the Basque Country UPV/EHU, 48013 Bilbao, Spain

Corresponding author: Imanol Mugarza (imugarza@ikerlan.es)

This work was supported in part by the Basque Government under the Elkartek 2017 Program (SEKUTEK project) under Grant KK-2017-00044.

ABSTRACT The Internet of Things (IoT) and Industrial Internet of Things (IIoT) trends, where high connectivity is envisioned, are giving rise to new applications, services, and paradigms, such as smart cities. Due to this connectivity and information sharing features, security, and privacy protection mechanisms need to be implemented, which may become obsolete at some future time. Software updates are, then, crucial. However, software updates requiring system shutdown and restarts might not be acceptable from the business and service point of view when high availability is demanded. In this paper, a mixed-criticality software architecture and design for a building energy management system, built upon the Cetratus runtime framework, is presented, where partitioning techniques are employed to ensure temporal and spatial isolation. Through this framework, software updates are dynamically accomplished, without the need for system shutdown and restarts. A live patching example is also presented, where customers privacy is enhanced by means of homomorphic encryption.

INDEX TERMS Smart city, smart energy, Cetratus, live updates, live patches, dynamic software updates, security, privacy, availability, partitioning.

I. INTRODUCTION

These days, information and communication technologies are being employed and integrated towards the accomplishment of interconnected and intelligent smart cities, with the overall aim of improving the quality of life of citizens. This might include the reduction of waste, resource consumption and/or the improvement of overall living costs. To this end, smart and advanced services are offered, e.g. in energy and transportation services. High interconnectivity among all sensing, storing, processing and analyzing devices is fundamental, a tendency enabled and promoted by the Internet of Things (IoT) and the Industrial Internet of Things (IIoT) technologies.

In the case of the energy sector, *Smart Energy* and *Smart Energy Systems* refer to the design and implementation of sustainable and cost-effective energy management strategies [1]. This topic is actually being analyzed and investigated in several research projects, such as in/by CITyFiED [2]. 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 greenhouse gas 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'' [2].

A smart grid platform which makes use of information and communication technologies is employed in CITyFiED for grid management solutions. Fig. 1 illustrates the adopted approach. The system is divided into different levels. First, a Building Energy Management System (BEMS) is defined, which gathers the energy flows of information from each of the buildings in the district. Secondly, the District Energy Management System (DEMS) monitors energy generation and distribution at district level. Electric car charging points are also installed in some car parking spaces. Data collected by the DEMS and BEMS system is transmitted to

The associate editor coordinating the review of this manuscript and approving it for publication was Mehedi Masud.

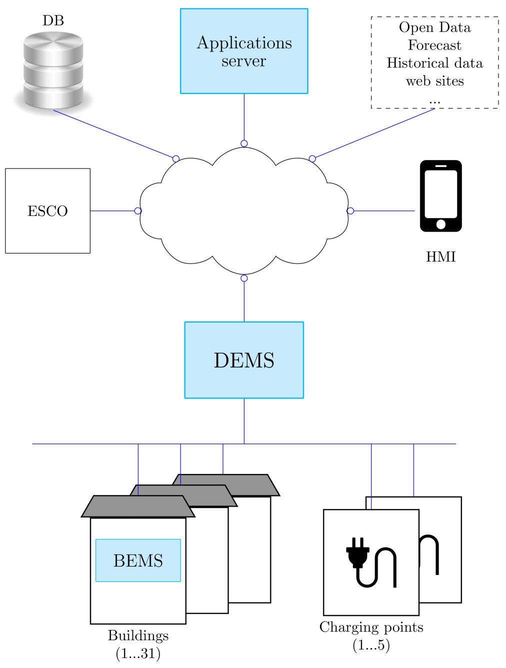

FIGURE 1. Smart grid and ICT system in CITyFiED [2].

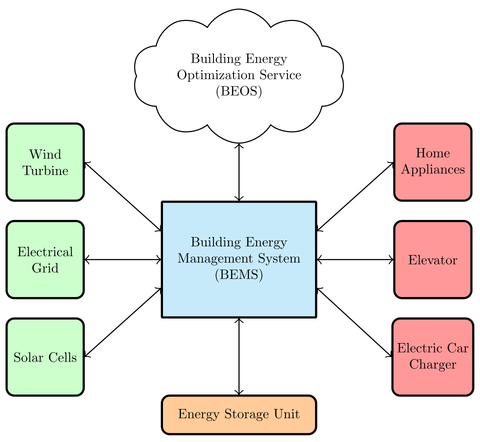

FIGURE 2. Building energy management system (BEMS).

the application server. For this purpose, third party information, e.g. weather forecasts or electricity tariffs published by the Energy Service Company (ESCO) are used. Moreover, a remote human-machine interface (HMI) provides some actual status, historical data and trending, as well as third-party information. The graphical user interface will provide a dashboard to display such data, as well as alerts and notifications in case of unnecessary waste of energy.

In this work, a smart building electrical energy management application is considered, consisting of the BEMS, and a Building Energy Optimization Service cloud application (BEOS). On the one hand, the BEMS is responsible for monitoring and controlling diverse energy-related facilities in a residential building. Firstly, various 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 on the roof of the building. Secondly, the supplied electrical power is used by various home appliances, the elevator and an electric car charger. The BEMS continuously measures the energy consumption of these devices. 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.

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 analyzed: the actual and expected electricity fees, and weather forecasts for renewable energy sources estimations. Fig. 2 shows the described smart energy application.

In the past, when applications had components with different criticality levels, such as security or safety, these components used to be implemented in separate computing platforms. This provided complete independence among application components, as faults at any part of the system would not be spread through the rest. However, it also led to a large number of devices and wiring with high cost in terms of equipment and maintenance. Nowadays, the trend in the design of many complex systems e.g. automotive, avionic, railway or industrial control systems, is to integrate application components with different criticality levels in the same execution platform. These are known as mixed-criticality systems [3] [4] and they have been deeply studied in many points of interest, from purely theoretical aspects e.g. scheduling and design issues, to basic implementation mechanisms [5].

Criticality is the term used to determine the degree of assurance that a certain component requires, and it includes many forms of dependability (availability and/or reliability), security (protection against attacks and/or intrusion confinement) and safety (fault containment) issues. It is typically used in the context of functional safety, and depending on the domain-specific standards the number of levels as well as their names may vary [6]. Functional safety refers to the identification of possible failures that may have serious consequences and to determine an acceptable rate of occurrence of these failures [7]. In IEC 61508 [8], which is a functional safety standard applicable to many kinds of industry, four discrete bands are defined to set the relative levels of risk-reduction of a certain component. They are called *Safety Integrity Level* (SIL): from SIL1, which is the lowest integrity level, to SIL4, which is the highest [7].

In this article, a mixed-criticality software architecture based on the *Cetratus* [9] runtime framework is proposed for the BEMS. This work provides a smart energy case study of the previously presented dynamic software updating approach. The presented scheme allows the dynamic update of application components. A live update case study is also 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.

This manuscript is organized as follows. After this introduction, the motivation for this work is given and the related work then exposed. The *Cetratus* runtime framework is presented first of all as the proposed solution. Next, the mixed-criticality software architecture and design of the BEMS are described. After that, a live update case study is illustrated as an example to validate the proposed solution. Finally, conclusions and future work are drawn up.

II. MOTIVATION

Due to the communication capabilities between the smart energy system and the cloud, security and privacy concerns arise. It has to be ensured that citizens information is securely sent and stored. The authenticity of such data shall be checked, since a malevolent attacker might actively eavesdrop such communications and examine, or even manipulate, private citizen data. Third-party services, e.g. the BEOS, would also be able to additionally access to this information, exposing customer living behaviors and habits. This information might be used maliciously [10]. Besides, consumers themselves could be interested in tampering with such metrics to decrease the electricity consumption bill, for example. If these concerns are not addressed, the smart city application or service may be vulnerable or susceptible to privacy data leakages [11].

At some point, in order to maintain the required security level, software updates will be necessary in order to fix security weaknesses and bugs. Software systems are inherently defective and software patches are unavoidable. Conventional software upgrade methods require to shut down the running systems and restart them from scratch. However, mission-critical and/or safety-related software systems require zero downtime executions. Any service interruption would lead to prohibitively expensive losses. Ordinary software updates might not be then feasible from the service point of view when high availability is requested. As stated by Khurana *et al.* [12], 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. In this case, any shutdown due to updating is unacceptable. However, if the security-related patches are not applied, the system will be exposed to attack. The reliability and trustworthiness of such a system will decrease as time goes by, as security protection measures could sooner or later become obsolete and be bypassed.

Fig. 3 illustrates the trust level on safety and security technologies during the operational period of a mixed-criticality system. As shown, in contrast to security, solid, stable and

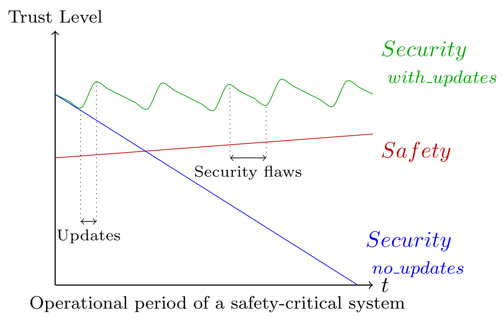

FIGURE 3. Safety & security trust levels through operational period.

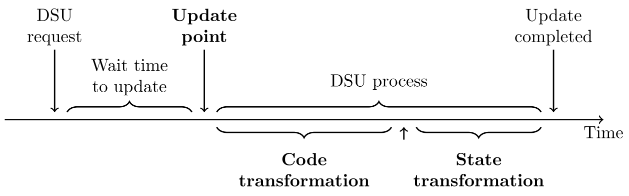

FIGURE 4. Dynamic software updating process time-line [9].

well-known technologies and methods are used in safety engineering. These technologies are further tested, verified and validated through time. This is the reason why the trust level in these technologies increases over time.

III. RELATED WORK

Hardware redundancy has been often employed to perform live system updates. To this end, a secondary hardware platform is utilized, where the new software version is loaded. The program state is then transmitted to it from the primary platform. After that, a role change is performed. The secondary platform is determined by primary one, and vice versa. At that moment, the hardware platform running the old software version can be switched off [13].

Dynamic Software Updating (DSU) techniques aim at upgrading or modifying computer programs while they are running without the need for a shutdown and restart. Hardware redundancy is not needed either. The execution of a computer program is defined as a tuple (P, δ), composed by the program code P and the current program state δ. On the one hand, the program code P contains a set of executable native instructions. On the other hand, the current program state δ includes all information related to the data structure of the computer program. It can include, in addition to the heap and the internal application data structures, the stack frames, program counters or the state stored by the operating system, such as file descriptors or already opened network connections. The dynamic software updating mechanism transforms the actual running program (P, δ) to a new version (P', δ'). To this end, code and state transformations are performed [13]–[15]. Fig. 4 illustrates the timeline process of a dynamic software update.

TABLE 1. Analyzed dynamic software updating systems.

Name	Target	Year	Reference
DLpop	Application	2001	[13]
OPUS		2005	[16]
DynSec		2013	[17]
POLUS		2007	[18]
UpStare		2009	[15] [19]
Ginseng		2008	[20]
Ekiden		2011	[21]
Kitsune		2012	[22]
DURTS	Real-time	2004	[23]
EmbedDSU		2011	[24]
Gracioli		2014	[25]
EcoDSU		2008	[26]
Seif-Real		2009	[27]
Wahler		2009	[28]
FASA		2014	[29] [30]

Table 1 shows a list of evaluated application-oriented and real-time systems oriented DSU systems. The analysis of these DSU mechanisms was performed by the same authors in [14]. In such work, DSU techniques targeting operating systems kernels are also investigated.

For the code transformation, trampolines are usually used through the insertion of dummy instructions. This technique was employed by OPUS, POLUS, UpStare, DURTS and Gracioli. In contrast, call indirections are handled in DLpop, Ginseng, EmbedDSU, Wahler and FASA. For this purpose, an indirection handling table is created, where direct function calls and returns are specified. In some cases, such as in DynSec and EcoDSU, the specific memory region is accessed and modified to replace the corresponding set of executable instructions. This technique is called binary rewriting. In DynSec, the code cache is flushed.

Concerning state transformations, three different methods can be used to perform such data updates: in-place, indirection and checkpointing. The first technique refers to overwriting the old program state information in place, directly on to the specific memory location where it is stored. Nevertheless, larger amounts of memory than the old program state might be needed by the new one. This technique is used by UpStare, Ginseng, Kitsune, EmbedDSU, EcoDSU. On the contrary, through indirections, the new program state is allocated in a new memory region and represented as a pointer of the underlying type to the new program. This method deals with the problem of requiring extra memory, in case the program data size grows. A memory management mechanism is then required. Indirection methods are employed by DLpop, OPUS and POLUS. Finally, checkpointing, also known as state migration or transferring, consists of packing and unpacking the program state from the old program to the new one. After the transmission, the state transformation functions are invoked. This technique is used by Ekiden, Kitsune, Wahler and FASA (Future Automation System Architecture).

All the DSU systems targeting compiled applications perform dynamic updates on top of an 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 provide real time features, which may be needed for an industrial control application. On the contrary, in the case of DSU systems designed for real-time, or embedded or industrial control systems, DURTS is the only DSU targeting a UNIX-like operating system, specifically, the RT-Mach operating system. Other operating systems were used on EmbedDSU and Gracioli. Wahler and FASA are operating system agnostics, although they shall be POSIX-compliant. They take advantage of the operating system utilities, so the DSU is transparent to the underlying operating system. Conversely, EcoDSU works purely on bare metal, without the support of an operating system. Real-time timing and tasking properties were analyzed in DURTS and Seif-Real [14].

None of the evaluated DSU system provide an industry compliant solution, although Wahler and FASA are the closest ones [14]. According to Wahler *et al.* [30], the biggest challenge of updating real-time systems is that they often need to be certified according to industry standards, e.g. the IEC 61508 [8]. If the software of a such system is updated, the whole system might be re-certified before it can be deployed in the field.

IV. PROPOSED SOLUTION

In this section, the mixed-criticality software architecture based on the *Cetratus* runtime framework is presented for the BEMS. Firstly, an overview of *Cetratus* is provided. The software architecture and defined application components are then described. In addition, a partitioning approach for the proposed software architecture is given, where temporal and spatial isolation are achieved.

A. CETRATUS

Cetratus is a framework enabling DSUs for safe and secure industrial control systems [9]. Its goal is to enable safe live updates of application components, allowing the adaptation to new requirements. This process is performed while running, without the need of powering off the system, hence, without degrading or losing availability. *Cetratus* is aligned with the industrial IEC 61508 [8] and IEC 62443 [31] standards and satisfies the internal test activity recommended by the *IEC 62443-2-3: Patch management in the IACS environment* technical document [31], [32].

An indirection handling table is created to manage which application component (and version) is actually executed. As far as state transformations are concerned, an application component state is defined by the programmer at the development phase. These accessible variables are then checkpointed and transformed (if required) for the new application component version [9]. Contrary to FASA [29] [30], whole system dynamic reconfiguration features are not present in *Cetratus*, since the use of such mechanisms is not advised for the development of safety-critical systems.

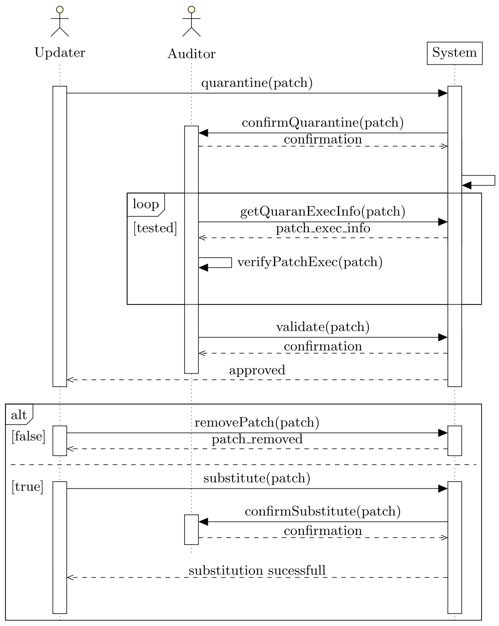

FIGURE 5. ***Cetratus*** **dynamic software update use case sequence diagram [9].**

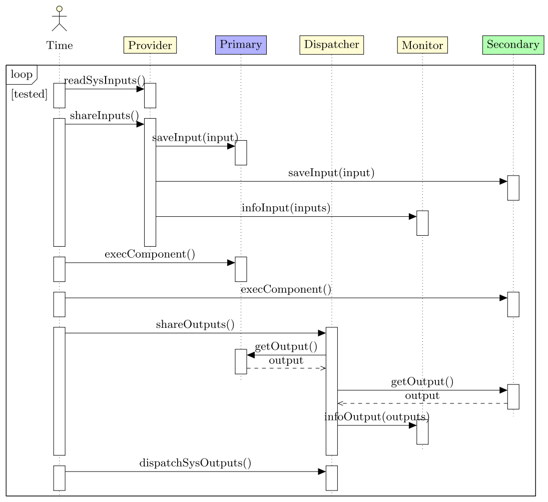

FIGURE 6. Quarantine-mode execution & monitoring for an application component [9].

Fig. 5 shows the *Cetratus* dynamic software update use case sequence diagram. Two different actors that interact with the system are defined: the *Updater* and the *Auditor*. The *Updater* is the system maintainer wishing to perform the upgrade. In contrast, the *Auditor* refers to the actor responsible for continuously gathering information of the patch execution and verifying its correctness, represented as a loop box in Fig. 3. The Auditor can reject the patch if it does not fulfil the expected behavior and/or performance. A corroboration from the Auditor is needed by the Updater to apply a given software update.

As illustrated in Fig. 5, a software update is firstly requested by the *Updater*. Whenever the *Auditor* endorses such an upgrade, the software patching process starts. At this moment, the new application is executed within the quarantine-mode until the decision is taken to replace the former application component. Through this isolation, protection against patch installation errors and/or new possible vulnerabilities or bugs is provided. The disturbance of unexpected error caused by dynamic updating will not damage the safety of the system. Meanwhile, the *Auditor* collects and verifies new application component execution evidences. If enough trustworthiness of the new software version is determined, the replacement is accomplished. For this purpose, *Auditor*'s confirmation is needed. Alternatively, the updater could also directly revoke the dynamic update.

Fig. 6 illustrates the quarantine-mode execution and monitoring scheme for a single application component. In the quarantine-mode, both the primary and secondary components can receive input and compute output. Nevertheless, only the outputs of the primary component are visible to the outside. System input and outputs are managed by the *Provider* and *Dispatcher* modules. A copy of application component input and output is also collected by the *Monitor*. This patch execution information is supplied to the *Auditor*.

As far as security is concerned, a digital signature is produced to ensure the integrity of the dynamic patch during its transmission to the target system, as required by the *SUM-4: Security Update Delivery* requirement in IEC 62443-4-1 [31]. Before proceeding with an update, the integrity and authenticity of the dynamic patch is verified by the runtime framework. A secure communication channel is also used to prevent any attacker obtaining the plain dynamic patch and reverse-engineering it. Any user interacting with the system shall also be previously identified and authenticated.

B. SOFTWARE ARCHITECTURE

The proposed software architecture for the BEMS is shown in Fig. 7. The software design is divided in two parts: the smart energy application and the *Cetratus* framework. At the bottom, *Cetratus* framework components are defined (shown in yellow). These elements are generic and reusable for any kind of application. At the top, application specific components are provided.

1) APPLICATION COMPONENTS

The application software running in the BEMS is divided among several application components, identified in Table 2. Wind and solar energy productions are measured by the C-WEM and C-HEM components. The C-HEM application components 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

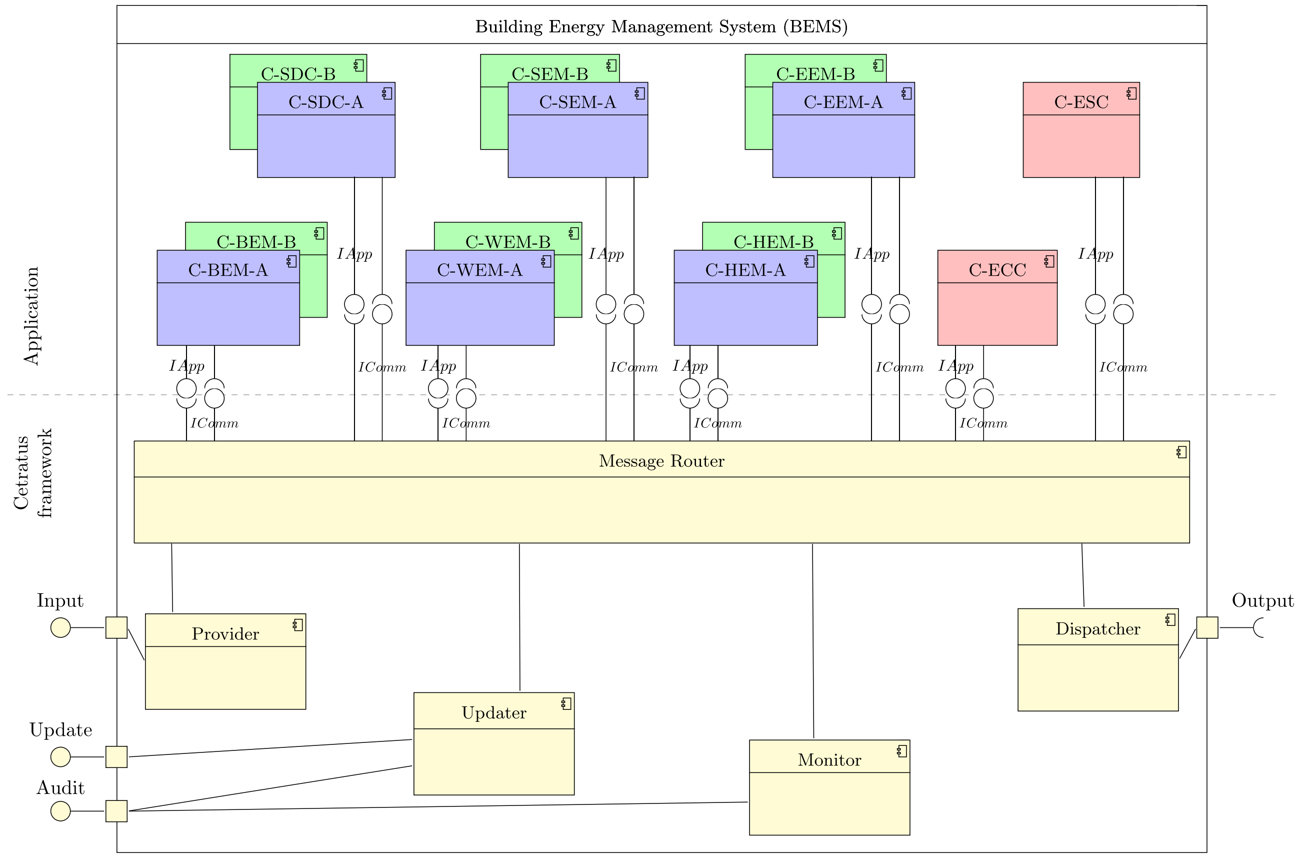

FIGURE 7. Software architecture of the building energy management system.

TABLE 2. Application components in BEMS.

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

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 safety standards.

The C-BEM is the overall building energy manager, and 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 from 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 (shown in blue and green), 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 [8]. Safety-related application components C-ECC and C-ESC (depicted in red) 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 [8]. Therefore, a secondary container is not required.

2) CETRATUS FRAMEWORK COMPONENTS

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 behavior, 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.

C. 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 unauthorized 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* [33] [34], 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* real-time operating system developed by *Green Hills Software* has been certified for security, safety and reliability domains, including IEC-61508 SIL3 [7]. Spatial partitioning is obtained by *Virtual Address Spaces* (VAS), which are protected memory regions of code and data that can only be reached by authorized processes. A *Partition Scheduler* is used to set a cyclic schedule of temporal partitions where different VAS-es 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 VAS 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 are 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 that are executed several times. Fig. 8 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 [35]. 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 of a higher degree of expandability without compromising its temporal restrictions.

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, 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 applications components. Thanks to partitioning, it is guaranteed that when running components in *Cetratus*, if any malfunction 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 Listing 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

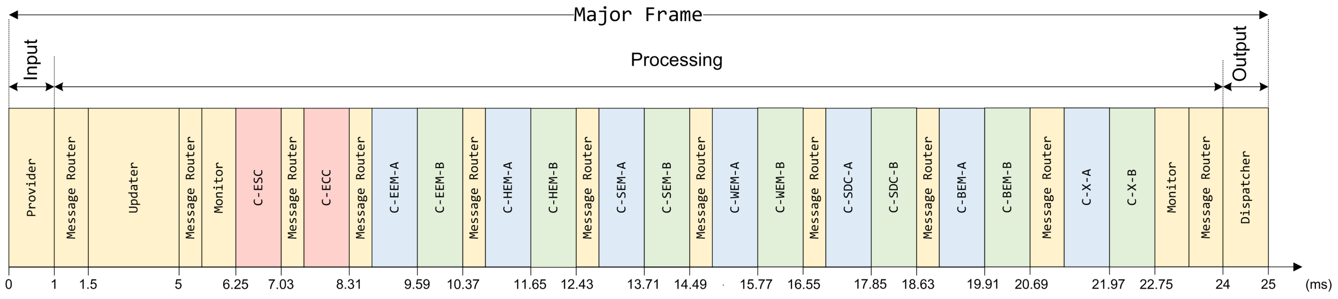

FIGURE 8. Temporal partition scheduling.

```
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 SDC_A
    AddressSpace CSDC_A #Secure Data Collector_A
    VAS
    Offset 17.07
    Exectime 0.78
  EndPartition

  Partition 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 1. ***Partition Scheduler*** **integration file.**

allocated to them to execute in their worst-case execution times.

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 next one scheduled 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.

V. VALIDATION

In order to validate the proposed mixed-criticality architecture, 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 [36]. 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 using this update.

Although a secure communication channel is used for the transmission of energy production, savings and consumption data, e.g. by means of Transport Layer Security (TLS) or any other encrypted and authenticated communication protocol, third-party services store and process smart energy data in clear text, so confidential information is completely accessible. This may compromise citizens' privacy if these third-party actors maliciously use such data; a well known example of this non-legitimate use is personal information selling. As stated previously, software updates are usually necessary to address any security and privacy issues that might be encountered during the operational period of the system. Assuming that the system has already been deployed and 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 [11]. Homomorphic cryptography is a cryptographic system in which computations can be performed on the ciphertext space. These operations are accomplished on encrypted data, from which the result of such calculations also remains encrypted. When decrypted, the solution matches the result of the computations as if they had been executed on plaintext data [36]. This approach protects private information contained in the transmitted data [37].

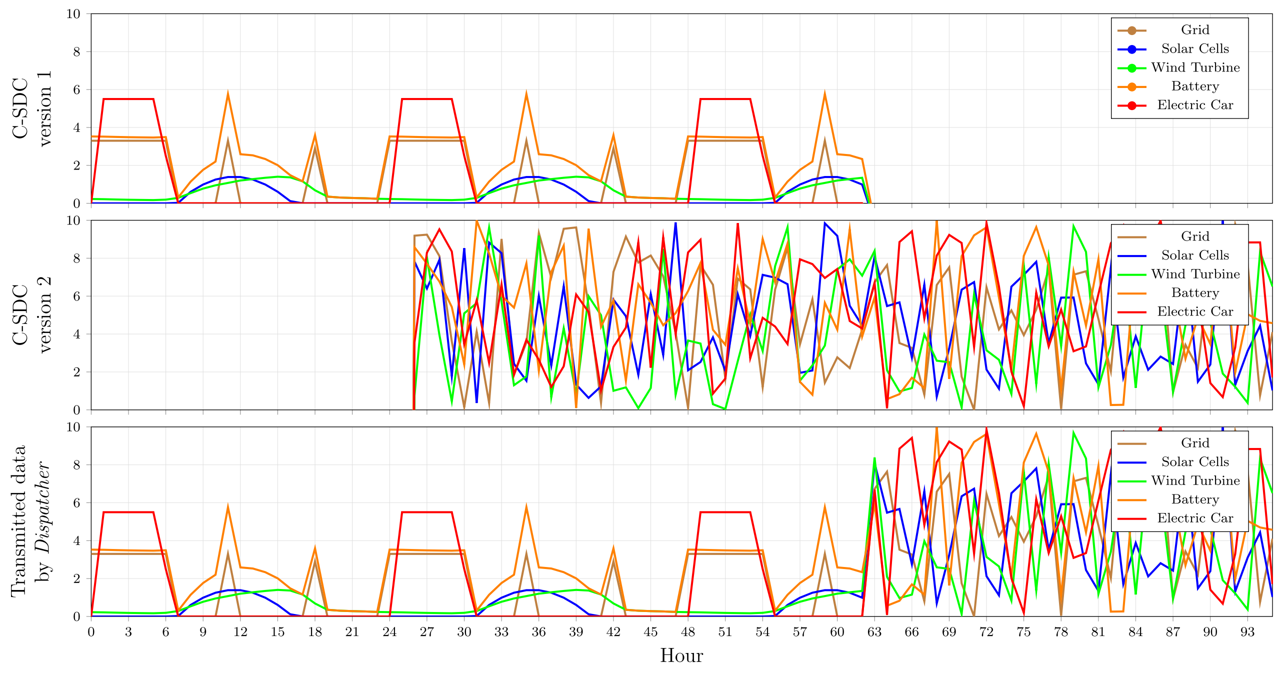

FIGURE 9. Energy production, savings and consumption data (in kWh) computed by both C-SDC application component versions and transmitted information to the cloud service.

A prototype of the presented BEMS has been developed, which is executed on a x86 industrial computer. The implemented *Cetratus* runtime framework is POSIX compatible and has been integrated over *INTEGRITY* real-time operating system. This implementation was originally linked and validated with Real-Time Linux [9]. Regarding dynamic software updates, features from the underlying *INTEGRITY* real-time operating system are employed.

Fig. 9 illustrates the outputs produced processed by both C-SDC application components, as well as 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 shown, 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 behavior patterns and private information might be obtained. 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 behavior of the new component verified. After this validation, a substitution of the former application component is performed. This step is accomplished at hour 63^{th} hour. The former C-SDC component is then stopped. 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.

Fig. 10 shows the system performance in terms of CPU and response times. At the top, the CPU time of both C-SDC-A and C-SDC-B application component versions during the live update process is shown in μs. As said before, C-SDC-A 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 this application component. 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 Fig. 10 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

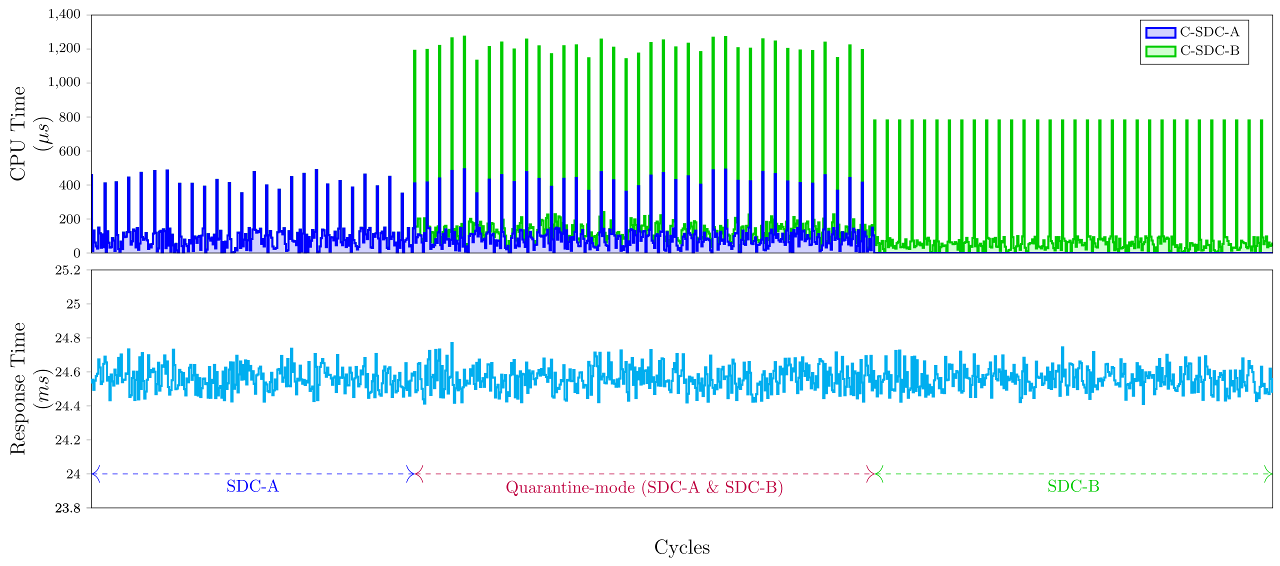

FIGURE 10. **CPU and system response times during live update (Integrity RTOS, executed on a x86 industrial computer).**

then capable of ensuring all the temporal requirements and constraints while both application component versions are being executed.

VI. CONCLUSIONS

In the current IoT and IIoT era, high inter-connectivity and information sharing is expected. These paradigms are enabling new applications and services. In the smart city domain, advanced and intelligent services might be offered to citizens, such as energy optimization services for cost-effective electrical energy usage practices. Therefore, security and privacy issues arise. During the system development stages, security protection measures shall be adopted. Nevertheless, these countermeasures might become obsolete after some time. Software updates are then necessary to mitigate those risks emerged during the operational period of the system. However, system shutdown and restarts are usually needed to perform such system upgrades. Software updates might not be then feasible from the service point of view in high availability smart city applications.

In this article, a mixed-criticality software architecture for high-availability smart energy applications is proposed. The software design of an Integrated Building Energy Controller is introduced, in which eight mixed-critical application components are defined. In order to guarantee a complete isolation among components necessary in such systems, a partitioning scheme based on *Integrity RTOS* has been proposed. The presented design is based on the *Cetratus* runtime framework, which enables the dynamic update of such application components. These upgrades are accomplished during runtime, without the need of shutdown and restart. A case scenario is provided, in which an additional security layer is introduced to the data collector agent. More specifically, the C-SDC application component is updated, where an homomorphic encryption algorithm is integrated to encrypt energy production, savings and consumption data.

As for future work, an access control scheme shall be used to authenticate, authorize and audit any user interacting with the system. In absence of such measures, an attacker could update the security-related components with dummy ones that disable previously adopted security countermeasures. As said before, there are two main approaches for partitioning: partitioning-enabled operating systems like the one presented in this work, and hypervisors. The presented solution in this article could be implemented following a hypervisor-based approach.

REFERENCES

[1] H. Lund, P. A. Østergaard, D. Connolly, and B. V. Mathiesen, "Smart energy and smart energy systems," *Energy*, vol. 137, pp. 556–565, Oct. 2017.
[2] D. Car, "Replicable and innovative future efficient districts and cities," in *Proc. ENERGY*, 2013, pp. 1–8.
[3] European Commission. Information Society and Media Directorate-General Unit G3/Computing Systems Research Objective, "Mixed criticality systems," Report from the Workshop on Mixed Criticality Systems, Brussels, Belgium, Feb. 2012.
[4] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Proc. 16th IEEE Real-Time Embedded Technol. Appl. Symp.*, Apr. 2010, pp. 13–22.
[5] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, no. 6, p. 82, Jan. 2018.
[6] R. Ernst and M. Di Natale, "Mixed criticality systems—A history of misconceptions?" *IEEE Des. Test*, vol. 33, no. 5, pp. 65–74, Oct. 2016.
[7] D. J. Smith and K. G. Simpson, *Functional Safety: A Straightforward Guide to Applying IEC 61508 and Related Standards*. Evanston, IL, USA: Routledge, 2004.
[8] *Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems*. Geneva, Switzerland: IEC, 2000.
[9] I. Mugarza, J. Parra, and E. Jacob, "Cetratus: Towards a live patching supported runtime for mixed-criticality safe and secure systems," in *Proc. IEEE 13th Int. Symp. Ind. Embedded Syst. (SIES)*, Jul. 2018, pp. 1–8.
[10] P. McDaniel and S. McLaughlin, "Security and privacy challenges in the smart grid," *IEEE Secur. Privacy*, vol. 7, no. 3, pp. 75–77, May/Jun. 2009.

[11] 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 Commun. Mag.*, vol. 55, no. 1, pp. 122–129, Jan. 2017.
[12] H. Khurana, M. Hadley, N. Lu, and D. A. Frincke, "Smart-grid security issues," *IEEE Secur. Privacy*, vol. 8, no. 1, pp. 81–85, Feb. 2010.
[13] M. Hicks, J. T. Moore, and S. Nettles, *Dynamic Software Updating*, vol. 36. New York, NY, USA: ACM, 2001.
[14] I. Mugarza, J. Parra, and E. Jacob, "Analysis of existing dynamic software updating techniques for safe and secure industrial control systems," *Int. J. Saf. Secur. Eng.*, vol. 8, no. 1, pp. 121–131, Jan. 2018.
[15] K. Makris, "Whole-program dynamic software updating," Ph.D. dissertation, Arizona State Univ., New York, NY, USA, 2009.
[16] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "Opus: Online patches and updates for security," in *Usenix Secur.*, vol. 5, p. 18, Jun. 2005.
[17] M. Payer, B. Bluntschli, and T. R. Gross, "Dynsec: On-the-fly code rewriting and repair," in *HotSWUp*, USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/hotswup13/workshop-program/presentation/payer
[18] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *Proc. 29th Int. Conf. Softw. Eng.*, Jun. 2007, pp. 271–281.
[19] K. Makris, "Upstare manual," Tech. Rep., 2012. [Online]. Available: http://files.mkgnu.net/files/upstare/UPSTARE_RELEASE_0-12-3/manual/
[20] I. G. Neamtiu, *Practical Dynamic Software Updating*. Ann Arbor, MI, USA: ProQuest, 2008.
[21] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, "State transfer for clear and efficient runtime updates," in *Proc. IEEE 27th Int. Conf. Data Eng. Workshops*, Apr. 2011, pp. 179–184.
[22] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for c," *ACM SIGPLAN*, vol. 47, no. 10, pp. 249–264, Oct. 2012.
[23] J. Montgomery, "A model for updating real-time applications," *Real-Time Syst.*, vol. 27, no. 2, pp. 169–189, 2004.
[24] A. C. Noubissi, J. Iguchi-Cartigny, and J.-L. Lanet, "Hot updates for java based smart cards," in *Proc. IEEE 27th Int. Conf. Data Eng. Workshops*, Apr. 2011, pp. 168–173.
[25] G. Gracioli and A. A. Fröhlich, "An operating system infrastructure for remote code update in deeply embedded systems," in *Proc. 1st Int. Workshop Hot Topics Softw. Upgrades*, Oct. 2008, p. 3.
[26] S. Kang, I. Chun, and W. Kim, "Dynamic software updating for cyber-physical systems," in *Proc. 18th IEEE Int. Symp. Consum. Electron.* Jun. 2014, pp. 1–3.
[27] H. Seifzadeh, A. A. P. Kazem, M. Kargahi, and A. Movaghar, "A method for dynamic software updating in real-time systems," in *Proc. 8th IEEE/ACIS Int. Conf. Comput. Inf. Sci.*, Jun. 2009, pp. 34–38.
[28] M. Wahler, S. Richter, and M. Oriol, "Dynamic software updates for real-time systems," in *Proc. 2nd Int. Workshop Hot Topics Softw. Upgrades*, Sep. 2009, p. 2.
[29] M. Wahler and M. Oriol, "Disruption-free software updates in automation systems," in *Proc. IEEE Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2014, pp. 1–8.
[30] M. Wahler, S. Richter, S. Kumar, and M. Oriol, "Non-disruptive large-scale component updates for real-time controllers," in *Proc. IEEE 27th Int. Conf. Data Eng. Workshops*, Aug. 2011, pp. 174–178.
[31] *Industrial Communication Networks - Network and System Security*. document IEC 62443, 2010.
[32] I. Mugarza, J. Parra, and E. Jacob, *Software Updates Safety Security Co-engineering*, Cham, Switzerland: Springer, 2017, pp. 199–210.
[33] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: A hypervisor for safety critical embedded systems," in *Proc. 11th Real-Time Linux Workshop*, 2009, pp. 263–272.
[34] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The xtratum approach," in *Proc. Eur. Dependable Comput. Conf.*, Apr. 2010, pp. 67–72.
[35] J. C. Palencia, M. G. Harbour, and J. J. Gutiérrez, and J. M. Rivas, "Response-time analysis in hierarchically-scheduled time-partitioned distributed systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 2017–2030, Jul. 2017.
[36] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library-seal v2. 1," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2017, pp. 3–18.
[37] S. Zhangy *et al.* (2018). "Genie: A secure, transparent sharing and services platform for genetic and health data." [Online]. Available: https://arxiv.org/abs/1811.01431

IMANOL MUGARZA received the B.S. degree in industrial electronic engineering from Mondragon University, Mondragon, Spain, in 2013, the B.S. degree in automation engineering from the University of Skövde, Skövde, Sweden, in 2013, and the M.S. degree in embedded and intelligent systems from Halmstad University, Halmstad, Sweden, in 2015. He is currently pursuing the Ph.D. degree with the IK4-IKERLAN Research Centre, Spain, in collaboration with the University of Basque Country, Bilbao, Spain.

In 2015, he joined IK4-IKERLAN, working as a Researcher within the dependable embedded systems area. He was a Visiting Researcher at LAAS-CNRS, Toulouse, France, within the Dependable Computing and Fault Tolerance (TSF) Group, in 2018. His research interest includes the design, development, and maintenance of mixed-criticality safe and secure systems.

ANDONI AMURRIO received the B.S. degree in telecommunications systems engineering from the University of the Basque Country, Bilbao, Spain, in 2014, and the M.S. degree in telecommunications engineering from the Autonomous University of Barcelona, Catalonia, Spain, in 2016. He is currently pursuing the Ph.D. degree in science and technology with IK4-IKERLAN Research Centre, Spain, in collaboration with the University of Cantabria, Cantabria, Spain.

In 2014, he was a Networks Technician for Telefónica Movistar, and from 2016 to 2018, he was a Test-Engineer Consultant at SEAT S.A. His research interest includes the development of scheduling and mapping algorithms, as well as timing analysis techniques, for mixed-criticality distributed real-time systems.

EKAIN AZKETA received the B.S. and M.S. degrees in telecommunications engineering from Mondragon University, Mondragon, Spain, in 2004 and 2007, respectively, and the Ph.D. degree in telecommunications engineering from University of the Basque Country, Bilbao, Spain, in 2013.

From 2007 to 2013, he was a Research Assistant with the Software Technologies Department, IK4-IKERLAN Technology Research Center, Mondragon. Since 2013, he has been a Researcher with IK4-IKERLAN Technology Research Center, where he is currently with the Dependable Software Team. His research interests include the optimization techniques for deployment and scheduling of distributed real-time systems, and architectures for mixed-criticality systems.

EDUARDO JACOB received the Ph.D. degree from ICT, University of the Basque Country, in 2001.

He spent a few years in the industry as a Network Manager first and an R&D Project Leader later. He has been the Head of the Department of Communications Engineering, University of the Basque Country. He also leads a research group, where he has directed several R&D projects at European level. He is also a Coordinator of the Smart Networks for Industry (SN4I) Industry 4.0 Experimental Network Facility. His research interests include the application of advanced networks (SDN/OpenFlow, NFV, and Slicing) to industrial applications, connectivity for ITS and security in distributed systems with recent results in authentication, and authorization in virtualized access networks and authorization in sensor networks.