Two Architecture Approaches for MILS Systems in Mobility Domains (Automobile, Railway and Avionik)

Daniel Adam
BMW Forschung und Technik
daniel.adam@bmw.de

Sergey Tverdyshev
SYSGO
sergey.tverdyshev@sysgo.com

Carsten Rolfes
Fraunhofer Institute for Applied and Integrated Security
carsten.rolfes@aisec.fraunhofer.de

Timo Sandmann
Karlsruhe Institute of Technology (KIT)
sandmann@kit.edu

ABSTRACT

Systems with mixed and independent levels of security and safety become more and more important in the future. In the German funded Bundesministerium für Bildung und Forschung (BMBF) research project ARAMiS (Automotive, Railway and Avionic Multicore Systems) different industry and scientific partners concerned on using multi-core processor for different security and safety critical use-cases.
This paper describes the motivation and use-cases behind the research actives in different mobility domains. Also two detailed descriptions and a comparison of two implementation for *Multiple Independent Levels of Security and Safety (MILS)* systems in mobility domains are included. In the end of the paper a outlook is given on potential further research activities on this research topic.

Keywords

MILS, Mobility, Safety, Security, Automotive, Railway, Avionics, FPGA

1. INTRODUCTION

Due to more and more complex functions on a Electronic Control Unit (ECU), the complexity of the software components and thus also the cost for the verification increases. Also the dependence between the software components must not be neglected, especially if they have different safety and security levels and run on the same ECU.
To attain a reduction of the system complexity and isolation of data despite increasing software scope, the MILS approach can be used. The basic idea of the MILS approach is that the critical parts of the system are small in terms of lines of code and also have a low complexity, so that they can be certified at high assurance levels. This is similar to the recommendation of the economy of mechanism from Saltzer and Schroeder for security systems [1, 2]. In this respect, a (complete) MILS system must fulfill the NEAT properties [3]:

- **N**on-bypassable: Policy enforcement functions cannot be circumvented.
- **E**valuatable: Policy enforcement functions are small enough and simple enough that proof of correctness is practical and affordable.
- **A**lways Invoked: Policy enforcement functions are invoked each and every time.
- **T**amperproof: Policy enforcement functions and the data that configures them cannot be modified without authorization.

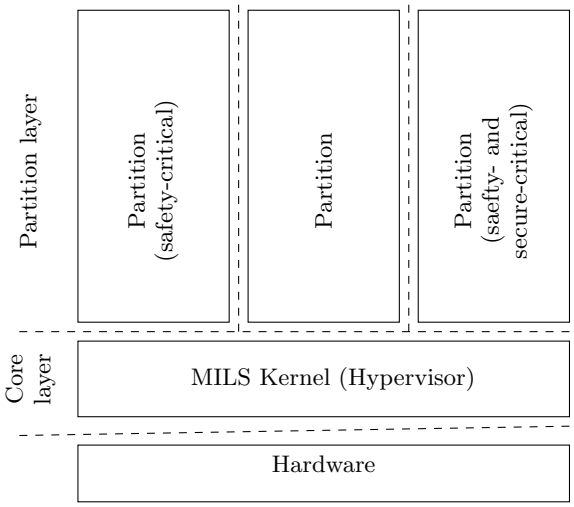

Figure 1: MILS architecture (Source: [3])

Basically a MILS architecture can be divided into two main (software) layers, see Figure 1. The kernel or core layer is responsible for the partitioning resources and maintains isolation across components. This core consists of several components, which implement and enforce the separation. This is the core component and an MLS is the most critical part of the architecture. In order to ensure independence, the kernel must be formally checked. The second main layer consists of the individual partitions, each of the partition corresponds an Single Level of Security and Safety (SLS) component [3].
To demonstrate the feasibility of multicore virtualization

and MILS approach as a key enabling technology for the centralization of functionality from different infotainment ECUs onto one single commercial off-the-shelf hardware platform, we built the Virtualized Car Telematics (VCT) computer [4]. It features an Intel-based multicore system along with a Xilinx Virtex-7 FPGA platform, both being connected to various input/output devices and other ECUs (e.g. via CAN and Ethernet). The complete hardware platform is built into the trunk of a BMW 320d touring car. The consolidation of functions into one single ECU includes safety relevant instrument cluster information as well as customer specific applications that are not qualified by the vehicle manufacturer. Those functions share dedicated hardware resources among several virtualized partitions.
Our paper is organized into 6 sections. Section 2 gives the motivation for the research activities and shows a current and future use-case. The following section 3 describes one architecture approach. Section 4 describes two realizations with examples. In section 5, we compare both realizations and discuss the differences. In the last section we give a outlook for potential future work and research topics in the field of MILS in the different mobility domains.

2. MOTIVATION

As already described in the introduction, were considered different use cases in the context of ARAMiS. In the following section we describe two use cases, in which the MILS pattern is a key solution. One use case is from the ARAMiS project and the second is from the current automotive research.

2.1 Use cases

According to Mark Weiser the ubiquitous computing and its follow-up approaches Cyber Physical Systems (CPS) and Internet of Things (IoT) is also becoming increasingly important in the automotive field [5]. Vehicles will soon be involved more deeply in the connected world, similar to smartphones at the moment. In addition to the automotive infotainment system, the classic driving systems will be more involved in the connected world, to enable a higher functional performance. The customer is expecting a similar functionality and flexibility like it is known of smartphones or tablet from the automotive infotainment systems, thus the customer demands are increasing rapidly. Especially when the previous driving time is available by self driving vehicle, the available infotainment options must be improved. To fulfillment of the customer demands can only be achieved by a widespread consumer operating systems, like Android or iOS, with a large range of apps. Due to to the usability the two areas entertainment and information can not be separated. These two areas must still be reconciled in the future, while the gap between their requirements is increasing.

While in the information sector stricter and higher security requirements are necessary due to self driving, in the entertainment sector a higher open awareness and flexibility is required due to networking and applications from the consumer sector. Future functions - regardless of whether its a driving function or from the infotainment area - have to satisfy the claim that these must be smarter. This smarter can be seen as a third processing stage. Thereby, in the future might exist three processing stages in an E/E system - reflex, conscious action, and cognition, similar to the human body. In order to achieve this intelligence in the newly introduced cognition stage mentioned earlier, bigger software components have to be used in which a protection is not always practicable. Therefore, these software components must be executed in a dedicated environment, which are separated from the other software environment to ensure security and safety according ISO 26262. It will become even more important to carry out a separation of different levels of safety and security [6].

2.2 Existing solutions

A study carried out a wide range of existing Android based in-vehicle infotainment (IVI) solutions. Many solutions were retrofitting solutions for the head unit, on which only one operating system instance is running. In most of this scenarios Android had a direct access to the Internet as well as to the electrical system of the vehicle. This leads to a entry point for attacks against the vehicle. An available Android IVI solution from AMG Mercedes, which must be ordered ex-factory, includes dedicated hardware for Android and a separate hardware firewall for protection against attacks of the electrical system of the car [7].

2.3 Research issue

The mentioned solution of AMG Mercedes is acceptable for exclusive cars, but not in production cars. Such a solution requires additional space in the car and must be considered in the construction phase of the car. Additional hardware and software interfaces are necessary, whichs leads to additional dependencies. But additional hardware means additional weight and this will increase energy consumption and fuel use. Therefore, it is necessary to find practical solutions, which allows variability without additional manufacturing and production costs. So a possible realization solutions for this is to use supervised AMP in order to have enough power to run two or more operating system instances. But it is important that the necessary peripheral hardware resources, such as Controller Area Network (CAN) or storage medium, can be shared. Here, the access method is important to ensure an operation without interference is possible. Especially when safety critical instances are executed. In the case of IVI a protection could be done, in which the IVI operating system is hardened or a walled garden is created around the IVI operating system.

This leads to loss benefits, which is gained through the use of consumer electronics (CE) operating system. Also update cycles would be thus extend, if adaptation must be performed on operating systems. Therefore, the protection without modification of the operating system must be done, which also allows quick update cycles, which especially in safety-critical updates is necessary.

3. ARCHITECTURE FOR MILS PLATFORMS

In the first part of this section we list automotive requirements for the MILS platforms. Then we give an overview of the different platforms. In the last section the platform is described more detailed.

3.1 Automotive Requirements on MILS Platform for Separation

The architecture of the MILS platform for both automotive systems has been driven by the use-case requirements. These requirements can be categorized into the following groups:

- Host Android for well-known HMI and Apps
- Execute embedded Linux for COTS drivers and software stacks
- Share graphics and QoS for graphics
- Provide connectivity for user equipment, e.g. USB
- Provide network connections, e.g. Ethernet
- Guarantee high assurance separation with gateways for security critical applications, e.g. payment, tachographhs
- Provide direct access to IO memory/devices, e.g. for latency critical devices
- Provide secure access to IO memory/devices, e.g. for shared devices or possible non-trusted drivers
- Execute legacy application, OSEK and/or old Autosar applications
- Enable ECU consolidation, i.e. running multitude of applications on the same ECU
- Support of different ASIL, i.e. support modular certification
- Dependable moderation of resources usage, e.g. CPU time, memory bus bandwidth
- Quick and Secure booting, e.g. secure boot and fast boot for critical applications
- Secure update, e.g. for partition and whole system
- Partition management, e.g. start, stop, restart
- System management and monitoring

3.2 MILS Platforms Overview

The MILS platforms used in both automotive systems consist of hardware and software parts. The hardware parts are based on COTS hardware (one x86 based and one ARM based) extended with domain specific devices prototyped on FPGA connected via common interfaces (ePCI, I2C). The software parts consists of separation kernels (SYSGO PikeOS, VxWorks), drivers for critical devices (e.g. PCI manager for DMA and TPM driver), non-critical device drivers, as well as Linux for COTS software stacks. A separation kernel takes control over the hardware, and thus, manages the hardware resources according the divined policy and provides the operation environment of separated partitions to execute system components.

3.3 PikeOS

PikeOS is a real-time operating system for safety and security critical applications [8, 9]. PikeOS is certified according standards DO-178B for avionics, IEC 61508 for safety in general, EN 50128 for railway and is under security certification Common Criteria. The PikeOS origin lies in the avionic area (e.g. Airbus A350, A400M), which is well-known for requiring highly robust components. Currently, PikeOS is used in different critical domains (e.g. automotive, railway, space, communication) it has highly modular and runs on a variety of hardware platforms.

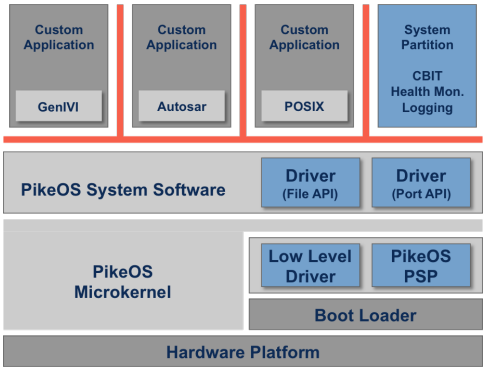

Figure 2: PikeOS architecture

Architecturally PikeOS consists of two major components: a micro-kernel and a virtualization layer (see Figure 2). The micro-kernel is very compact and provides the very basic functionality inspired by the ideas of Liedtke [10]. The virtualization layer is implemented on the top of the micro-kernel and provides separated execution partitions, also known as virtual machines, for user applications. User applications run in the isolated partitions which can be "personalized" with APIs, e.g. POSIX, Autosar, Linux/GenIVI etc. The scheduler of PikeOS is a multi-core real-time time-triggered scheduling with support of integration of events-triggered threads.

One of the critical issue on multicore is usage of common hardware resources such as memory bus and control of resulting interferences. We use PikeOS support to avoid interferences between critical SW components by utilizing space and time partitioning for multicore [11]. Figures 3 shows a typical multicore setup. In this configuration there are 4 cores (A-D) and 4 resource partitions (RP1-RP4). There are also 3 time partitions. In the same way as a resource partition "contains" the software access to memory and devices etc, so time partitions can be thought of as a container which dictates when software can execute. Using the above methods, software can be configured to not only run on a particular processing core but also at a particular time. Furthermore, using time partitions for multicore one can restrict what runs at the same time on different cores. This directly addresses one of the concerns raised earlier regarding interference patterns between cores. This architecture allows to control amount of interferences on the shared hardware through the platform, and hence, provide more guarantees for safety (i.e. WCET) and security (timing cover channels) critical applications.

Another important abilities for any separation kernel is to support precise security domains and integration strategy for critical components, e.g. critical device drivers. Thus, MILS platforms shall provide dependable support (e.g. implemented as a driver) to manage resources and functionality provided by these devices according the system level security policy. In this paper we have two such devices: PCI complex and TPM connected via i2c bus. PikeOS provides a PCI manager which is implemented as a separated SW component despite its direct access to the PCI complex. Similarly, TPM integration is also implemented as a isolated SW component which allows several partitions (i.e. virtual machine) to access it on their dedicated interfaces. Thus,

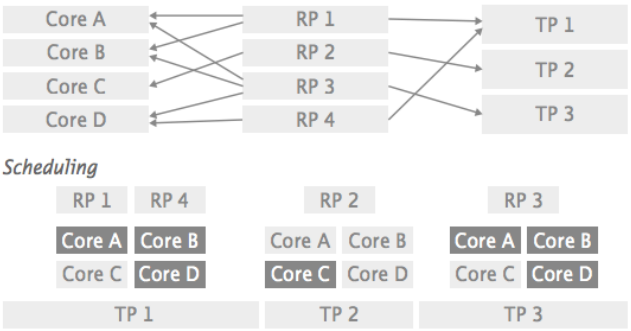

Figure 3: Space and Time partitioning with PikeOS

this allows creation of a virtual TPM device (see Figure 9).

4. REALIZATION

In the following two subsections, two different implementations are described which develops and be used within the ARAMiS project.

On platform A which is based on the Wind River hypervisor, a hardware-based virtualization approach for shared coprocessors is used. The second implementation on platform B based on the Sysgo hypervisor, a software-based virtualization approach for a hardware security module is used instead.

4.1 Hardware-based Virtualization Approach

In multicore systems, competing parallel accesses to shared resources like interconnects, memory or coprocessors can lead to collisions as only one of the accessing cores can be successful. All other cores get at least delayed which can lead to missing deadlines of applications. Therefore the design of safety critical systems has to achieve determinism as well as segregation in time and space.

In order to address this challenge and to support determinism for shared resource usage we developed a hardware based scheduling and interface approach for coprocessors that is able to guarantee Quality of Service (QoS) in terms of execution time, throughput and latency for hard real-time systems. It is a hardware extension of the coprocessor and can be seen as a wrapping layer around the coprocessor functionality which is transparent for the applications, meaning that no additional control steps besides the initial configuration have to be integrated in existing applications.

A use case for a shared resource in a multicore system, are cryptographic coprocessors. These coprocessors, having a much better latency and performance-to-power-consumption ratio than general-purpose processors, are required for cryptographic secured communication of embedded systems with low latency such as in Vehicle-to-X communication scenarios (e.g. ECDSA [12]). They can furthermore be used in a Trusted Boot/Execution scenario, where software codes have to be cryptographically verified before they can be started.

These use cases also show that access requests may be from very different nature: Vehicle-to-X communication messages are sent and received periodically requiring a hard real-time behavior but having a small data amount, whereas the verification of software codes has rather large data amounts but occurs only once (Trusted Boot) or sporadic (Trusted Execution) and tends to be soft or not real-time.

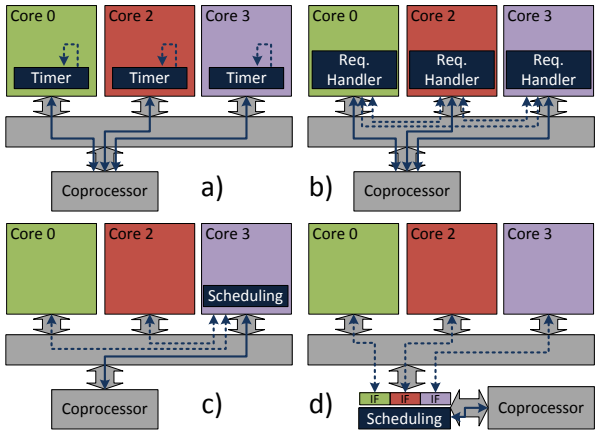

Figure 4: Variants of Resource Sharing Mechanisms for Multicore Systems

Multicore Resource Sharing

Sharing of resources like coprocessors can be achieved through different mechanisms. Temporal segregation is the major challenge as typically a software task requiring the service of a coprocessor would send a request to the coprocessor, including pointers locating required data in the main memory. The coprocessor then fetches those data from the memory using the DMA engine, calculates the results and then writes them back to the memory using the DMA engine. If another task requests the service of the coprocessor during that time, it has to wait an unknown amount of time for the execution to finish. That is unacceptable for the determinism and real-time behavior of coprocessor requests respecting their software tasks.

Different mechanisms for the sharing of a coprocessor are exemplarily shown in Fig. 4. All four systems consist of three cores (in this context equal to partitions), which try to gain access to one shared coprocessing resource over a shared interconnect. Here an asymmetric processing approach is assumed, where each core has its own independent task scheduling. For simplicity this figure abstracts from operating systems and virtualization solutions.

Fig. 4 a) shows a time based solution for shared resource accesses. Each core has a timer, that indicates based on a predefined schedule, when access to the resource is granted. This means that the partitioning and assignment of fixed time slices has been done during design time. The major disadvantage is the poor efficiency due to empty slots, whenever it is not used by the corresponding core.

In Fig. 4 b) the coprocessor resource sharing is not time but request based. Whenever a partition has a request it checks the coprocessors availability in coordination with the other cores. If the resource is free it gets access. Major disadvantage here is the full blocking of the resource by any of the accesses. Additionally as the scheme is completely distributed it does not allow to enforce priority rules. Also all partitions need the same safety and trust level.

Fig. 4 c) proposes a centralized approach where one partition schedules the requests to the shared resource. This approach supports spatial and temporal isolation but has the disadvantage of additional latency. Such an emulation

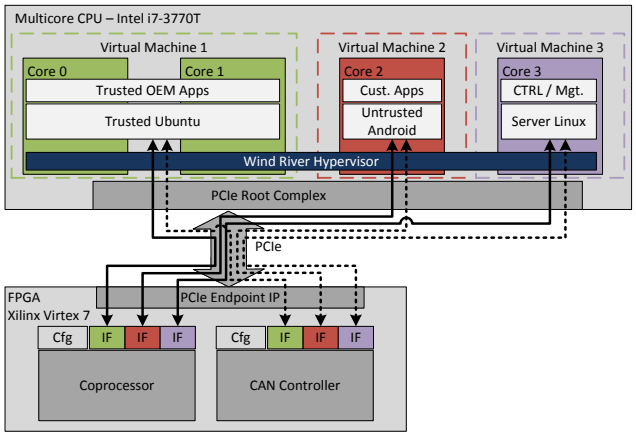

Figure 5: System Architecture of the VCT Intel-based Demonstrator Computer

approach of software-based virtualization for hardware resources sharing can not fulfill the strict performance and real-time requirements of many safety-critical systems.

Fig. 4 d) depicts the approach, which is described below. Basically it is the same as in c) but moves the shared resource scheduling close to that resource. This way a tight coupling of the resource and the request scheduling can be achieved. Moreover the scheduling can be implemented in hardware, which is the key for fine grain and real-time scheduling decisions. The scheduling extension also provides several interfaces for the different cores/partitions. This way spatial segregation can be easily achieved by means of the existing MMU infrastructure. No additional interaction among the cores for resource access is necessary.

Virtualized Car Telematics Computer

Fig. 5 shows the system architecture of the VCT computer. It consists of four cores, namely *Core 0* to *Core 3* and three *Virtual Machines*, which can be seen as containers for applications and run time infrastructure, running on top of the Wind River hypervisor. *Core 0* and *Core 1* are jointly used by *Virtual Machine 1* that executes one operation system providing the infrastructure for applications on these two cores. Similarly *Virtual Machine 2* and *Virtual Machine 3* have their own operating systems and applications. The FPGA is used for the hardware implementation of a shared devices, a cryptographic coprocessor for Vehicle-to-X communication. The virtual machines are segregated from each other, so that failures within one can not spread to others. While the trusted VM based on Ubuntu contains only software from the vehicle manufacturer which is supposed to be safety critical and free from malicious code, an untrusted VM running Android is available for the integration of arbitrary user applications. Furthermore, there is a server VM running a Linux distribution which provides services used by the remaining VMs and additional management tasks.

Virtualization Technology for Shared Resources

Following the trend towards virtualization, the PCI-SIG released the SR-IOV specification [13] as an extension to the PCIe standard. SR-IOV refines the notion of a PCI function, distinguishing between physical and virtual functions (PF/VF). A PF corresponds to an existing PCI function in a PCIe system without SR-IOV, while a light-weight VF is meant to be accessed directly from virtual machines. Each VF is assigned to a PF in a manner that the same PCI configuration resources are shared between a PF and its VFs. Other function-specific hardware resources (data buffers, work queues, etc.) are available exclusively to each VF and are accessible from a virtual machine without intervention of software.

With Xilinx's Virtex-7 FPGA series, support for SR-IOV has been recently added to their latest PCIe Endpoint IP block. SR-IOV has been designed to be able to work with existing system-level virtualization supports (e.g Intel Virtualization Technology for Directed I/O [14]). The cooperation between SR-IOV and these technologies enables the efficient spatial segregation of PCIe accesses on the path from the CPU into PCIe devices using the low overhead of direct device assignment as shown in [15]. This makes SR-IOV the favored choice for virtualization of PCIe devices in an embedded environment.

Coprocessor Architecture

The principal approach is to make the coprocessor look like several coprocessors for the remaining part of the system. This is similar to device emulation in virtualization concepts that comes along with the aforementioned performance and latency limitations (similar to Fig. 4 b). The proposed coprocessor architecture consists of the coprocessor, a hardware extension for virtualization support and a hardware scheduler to achieve the deterministic, segregated and efficient management and processing of coprocessor requests in multicore safety-critical systems. Within the hardware extension layer, there are one configuration interface and several access interfaces supporting the system partitioning concept from above: access requests from cores as well as data exchanges corresponding to different partitions are routed to different interfaces. This way spatial segregation of partitions is guaranteed.

With these access interfaces, each VM can be given exclusive access to its "own" accelerator reducing hypervisor overheads. To establish an as generic as possible interface for shared coprocessors, the connecting infrastructure is designed and implemented in a highly flexible manner in order to support a wide variety and easy replacement of coprocessors. Our approach covers the adaption to a standard PCIe interface on one side and a generic AXI interconnect [16] on the other side. Using an AXI interconnect allows to connect a wide range of components with different performance and throughput requirements.

The requests will be scheduled for execution on the coprocessor according to a defined configuration by a hardware scheduler. The configuration interface is solely accessible under specific circumstances (e.g. at boot time) by a trusted partition or the hypervisor.

Virtualized Coprocessor Interface

To couple the cryptographic coprocessor used in the VCT demonstrator with the host system, we developed a flexible coprocessor interface infrastructure for virtualized systems using SR-IOV and fulfilling the requirements stated above. These generic infrastructure implementation is described in more detail afterwards.

We implemented our approach on a Xilinx VC709 FPGA board [17] which supports the PCIe interface by Xilinx's

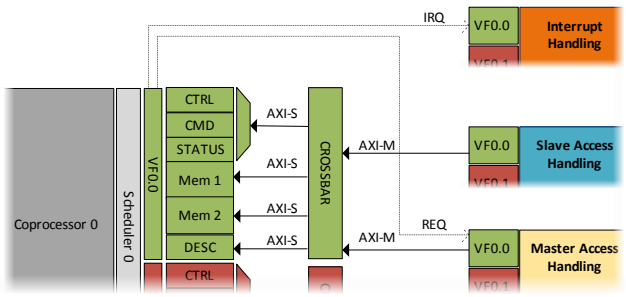

Figure 6: Structural Description of Coprocessor Interface

PCIe Integrated Gen3 Endpoint IP core [18]. The PCIe interface offers the communication infrastructure for all transfers of all SR-IOV virtual and physical functions. Our connected infrastructure architecture allows coprocessors to maintain an interface independent of the PCIe endpoint, thereby decoupling the coprocessor from the endpoint's implementation and PCIe communication details.

Besides the *Virtex-7 PCIe Endpoint*, the modular design of our developed infrastructure consists of four major parts (see Fig. 6): An *Adapter Control* and *Slave Access Handling* module for host initiated accesses, a *Master Access Handling* module for coprocessor initiated accesses, an *Interrupt Handling* module and the shared coprocessors *Coprocessors [0,...,N]* with each of them providing the function interfaces motivated by PCIe SR-IOV.

The *Virtex-7 PCIe Endpoint* block provides the PCIe link to the host system and SR-IOV with up to 2 physical and 6 virtual functions. It provides two 128 bit wide AXI4-stream based interfaces for host accesses and DMA request. Furthermore it provides interrupt and configuration interfaces.

The generic interface model for the connected coprocessors is shown in green in Fig. 6 using the example of the first virtual function (VF0). It consists of a set of registers for control purpose as well as additional coprocessor local memory for internal data and DMA descriptors. All interface components are available through AXI interconnects from two different sources each: the *Slave Access Handling* and the *Master Access Handling* module. Therefore each AXI crossbar contains two master inputs (AXI-M). Both bus masters are able to access the virtual function control module as well as virtual function local memory. For triggering interrupts or requesting DMA operations, every virtual function of the interface has dedicated signals.

The whole interface is replicated N times for a coprocessor shared between N virtual functions to provide segregation in space (distinct address spaces of virtual functions) and time (scheduling the accesses accordingly to the accessing function). Incoming requests for coprocessors will be scheduled for execution on the coprocessor according to a defined configuration by a hardware scheduler that is part of the coprocessor. In an analogous manner DMA transfers are scheduled by the DMA engine of our interface architecture for shared coprocessors [19] to support bandwidth management as required for safety-critical systems.

4.2 Software-based Virtualization Approach

To enhance the security of embedded systems, Hardware Security Modules (HSMs) are used. These modules offer secure storage of keys, integrity checks of software, and implementations of cryptographic algorithms. This crypto functions can be used to establish secure connections to OEMs backend systems or to ensure the integrity of software running on ECUs. But today's HSM designs lack the support of separation and virtualization, though. Current approaches to circumvent this issue are based on software emulation of the full or parts of the HSM. In spite of having notable advantages, system virtualization imposes software complexity and the necessity to virtualize the underlying Hardware Security Module (HSM) as well. In contrast to existing Trusted Platform Module(TPM) solutions that internally only multiplex several physical TPMs, multiple contexts shall be handled by a single TPM that is capable of securely swapping contexts. The HSM is implemented on a FPGA platform with a soft-core processor. The TPM context data for every specific virtual machine is initialized and stored encrypted in an off-chip embedded flash as the on-chip ROM of TPM is limited and extremely costly. Additionally, a scheduler is designed to reduce the number of context switches. In this section we present the integration results of an FPGA based HSM prototype with a real world embedded multicore system from Freescale. Both systems are connected via an I2C interface.

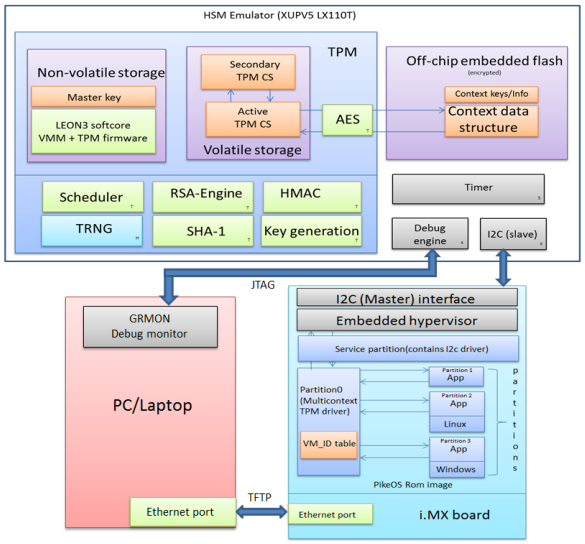

Figure 7: Detailed Architecture

Integration Architecture

The complete architecture of a multi-context TPM demonstrator is shown in Fig. 7. The HSM is implemented on a XILINX Virtex5 FPGA platform with LEON3 open source soft cores (SPARC V8 instruction set). The HSM in this case is a TPM capable of handling multiple contexts. The virtualized TPM firmware resides on the flash memory of the LEON3 soft-core. The i.MX6 Quad application processor is used to implement the virtual machine environment. The HSM and the virtualized application processor are connected via I2C interface. The HSM acts as a I2C slave while the multicore application processor acts as I2C master. The TPM context data for every specific virtual machine is initialized and stored in the external embedded flash as the on-chip TPM ROM is limited. The TPM context data is

highly sensitive and hence approaches are designed for the secure storage of data in the off-chip memory.

An additional layer would be needed to incorporate the multi-context capability into TPM 1.2 specification. The Virtual Machine Manager (VMM), an additional layer which performs the secure context switching and scheduling is therefore implemented in firmware, which resides on the LEON3 soft-core processor. The off-chip embedded flash holds the permanent data of every context and the data is encrypted using an AES software module before leaving the TPM. A scheduler is implemented which minimized the number of context switches to improve the performance. In addition to VMM, the general purpose timer modules are configured to accurately measure the time taken for context switching and for encrypting/decrypting context data. The debug engine is used for connecting the board to the GRMON debugger with the help of JTAG connector. The TPM emulator code resides on the flash memory of the LEON3 soft-core and makes use of the hardware accelerators for True Random Number Generation(TRNG). Except the TRNG, all the other cryptographic modules and the VMM is implemented in the firmware.

The application processor in our system is a virtualized embedded system with PikeOS [20] Operating System. PikeOS is a microkernel-based real-time operating system which is targeted at safety and security critical embedded systems. It provides a partitioned environment for multiple operating systems with different design goals and security requirements to coexist in a single machine. The architecture of PikeOS [20], which implements the hypervisor and the virtual machines on the i.MX6 board.

Apart from the service partition and the application partitions, we need to have a dedicated partition which acts as the multi-context TPM driver. This partition holds the VM_ID table, which maps the virtual machine with a four byte unique identifier. The other partitions are application partitions, which might either run a secure application or an Operating System such as embedded linux [21]. Usually, the hypervisor will implement the VM_ID mapping as the VM_ID table needs to be highly secure. As the hypervisor is a part of the PikeOS kernel [22] which cannot be modified, the functionality is included in a separate partition. It is the only partition which directly communicates with the multi-context TPM and is responsible for framing the multi-context commands. It communicates with all the other application partitions using the inter-process communication and shared memory concept described in [23]. It collects the TPM commands from the shared memory of a specific partition and attaches the VM_ID as defined in the VM_ID table. After framing the multi-context TPM command, it is stored in a buffer, which is then transmitted to the multi-context TPM module for execution.

Hardware Security Module

The TPM 1.2 standard from 2003 does not include any capabilities to store and load contexts, which would be required for context switching. In contrast to existing TPM solutions that internally only multiplex several physical TPMs, multiple contexts shall be handled by a single TPM that is capable of securely swapping contexts. All the limitations of the existing multi-context TPM architectures need to be taken into consideration as well. One of the central issued is to investigate the best strategies to reduce the extra time for

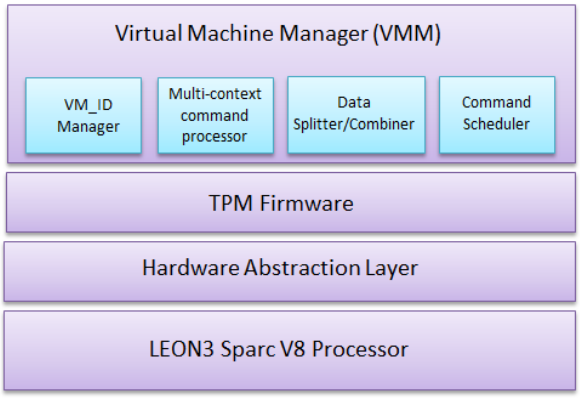

Figure 8: TPM Firmware Architecture

fetching and storing the data during each switching event. Furthermore, a scheduler will be implemented to avoid unintentional switching activity. The multi-context TPM functionality is implemented using a softcore processor, which communicates to a multicore application processor.

The implemented multi-context firmware architecture and the different layers of the software are illustrated in Fig. 8. Immediately above the LEON3 processor hardware, we have the Hardware Abstraction Layer which contains a set of software routines which emulate some platform-specific details, giving programs direct access to the hardware resources. The hardware peripherals can be controlled and accessed by a set of Special Function Registers (SFR) which include control register, status register, interrupt control registers and data registers. The peripheral configurations such as general purpose timers and external memory controller, the PROM and I2C drivers are implemented in this layer.

The TPM firmware is implemented using the Application Peripheral Interfaces (API) provided by Hardware Abstraction Layer. It is based on the software based TPM emulator code from [24], which is ported to LEON3 processor. The TPM firmware implements execution engine and cryptographic modules. A hardware block for True Random Number Generation (TRNG) was already implemented at AISEC and integrated into the design. In the initial prototype implementation, it was decided to use the TPM firmware as such and not to modify the TPM firmware to avoid complexities.

The Virtual Machine Manager (VMM) is the topmost layer in abstraction, which implements the features to support multiple contexts. This layer uses the underlying TPM firmware and the APIs provided by the Hardware Abstraction Layer (HAL). The static context initialization approach is adopted in the implementation and testing of the prototype would be less complex. Provisions has been made to store six different contexts and each context is assigned a specific VM_ID. The VMM layer consists of sub-modules such as VM_ID manager and address mapper, data splitter/combiner, multi-context TPM command handler and a dynamic scheduler.

Multicore Platform i.MX6

The implementation of the virtualized application system is shown in Fig. 9. A physical I2C bus connects the i.MX board which acts as the I2C master and the multi-context TPM

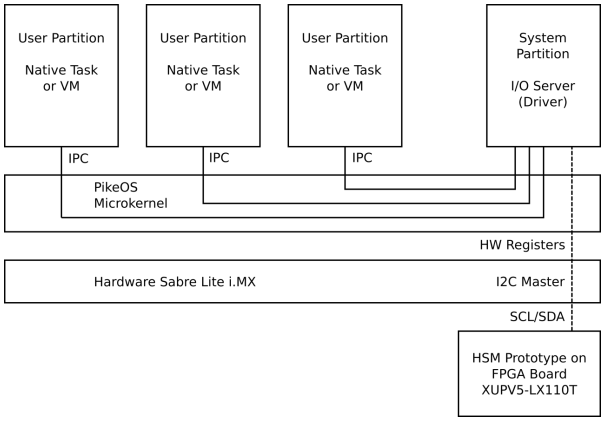

Figure 9: Block Diagram of a Virtualised TPM System

which acts as the I2C slave. The multi-context TPM driver is implemented on one partition (Partition 0) of the virtualized system. It also implements routines for sending and receiving multi-context TPM commands. As the PikeOS hypervisor is a part of the kernel which cannot be modified, the VM_ID table is maintained in a separate dedicated partition. The TPM driver partition make use of the underlying I2C driver functions on the kernel level. The partition consists of the VM_ID table, which provides a unique ID to every VM which needs an access to the TPM. The driver partition is the only partition which is interacting with the TPM. All the other partitions interact with this main driver partition, exploiting the shared-memory and Inter-Process Communication concepts.

5. DISCUSSION

Requirement	Platform A	Platform B
Latency	low	medium
Bandwidth	high	low
Predictability	available	available
Flexibility	low	medium
Development cost	medium	low
Interface	high speed (PCIe)	low speed, robust (I2C, SPI)
Legacy code	supported	not supported

Table 1: Comparison of two Implementations

Comparing the two virtualization approaches for shared resources (coprocessors on platform A and a hardware security module on platform B), different requirements lead to different approaches. The requirements and key features of both implementations are compared in Table 1.

For the hardware-based virtualization approach for coprocessors, the key requirements are a low latency and high bandwidth coupling of the coprocessor with support for DMA operations as well as deterministic access times even in the case of a concurrent usage on a multicore architecture. Therefore out implementation provides an access latency of about 530 ns and a total DMA data transfer bandwidth of 2.6 GiB/s which can be deterministically allocated to different partitions as presented in [19] at the cost of additional hardware resources. Furthermore this approach is based on the PCIe SR-IOV standard [13] and therefore independent of the operating system and base software stacks running on the system to provide flexibility and independence from the platform and software architecture. In this way (legacy) applications and software stacks using the coprocessor resources don't need any modification when integrated on a virtualized multicore platform. However the host system architecture has to support the PCIe SR-IOV standard. Additionally our hardware scheduling and DMA bandwidth monitoring allows a predictable behavior independent from the CPU utilization of the host system.

The implementation of platform B: PikeOS was mainly driven by the limited resources of low cost embedded systems used in many of the electronic control units inside a car. The development cost must be very low and the lifetime is high compared to consumer products. On the other hand the use of well established interfaces, that are robust have a low wire count, is intended. Therefor, the bandwidth and the latency of the interface is not that important. But the limited bandwidth of existing bus structures like I2C, SPI, and LPC will become a problem, if more and more data needs to be encrypted or hashed.

6. FUTURE WORK AND RESEARCH

One of the future work is to involve hardware manufactures to provide better knobs to separate information flows. This will allow separation kernel to provide tuneable control (e.g. side effects via common hardware infrastructure) and increase performance (e.g. virtual functions in hardware).

One of the interesting issues is how to security certify the presented systems in a modular way, e.g. we are looking on how to build up system level assurance leveraging guarantees provided by the MILS platform. Another big open questions is how much hardware manufactures can contribute and how much is needed from them to provide strong guarantees for the MILS platform. Here we are looking for MILS eco-system (e.g. for the beginning consisting of basic elements such as hardware, separation kernel, and some critical services for applications) which will provide guidelines how that composite assurance can be derived from the MILS architecture.

The hardware-based virtualization approach is currently being ported to the ARM-based Zynq architecture from Xilinx using an AXI bus interfaces instead of PCIe used on the Intel platform. This enables the support of our hardware virtualization concept for coprocessors with a low latency interconnect on a typical embedded system architecture. The software based approach can also benefit from the onchip interconnect when integrating of the cryptographic module directly onto the application processor.

Acknowledgment

This work was funded within the project ARAMiS by the German Federal Ministry for Education and Research with the funding IDs 01IS11035. The responsibility for the content remains with the authors.

7. ADDITIONAL AUTHORS

Additional authors: Steffen Baehr (Karlsruhe Institute of Technology, `steffen.baehr@kit.edu`), Oliver Sander (Karlsruhe Institute of Technology, `sander@kit.edu`), Uwe Baum-

garten (Technische Universität München, baumgarten@tum.de) and Juergen Becker (Karlsruhe Institute of Technology, becker@kit.edu).

8. REFERENCES

[1] P. W. O. Jim Alves-Foss, "The MILS architecture for high-assurance embedded systems." *IJES*, vol. 2, no. 3/4, pp. 239–247, 2006.

[2] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278 – 1308, Sep. 1975.

[3] J. Delange, L. Pautet, and F. Kordon, "Design, implementation and verification of MILS systems," *Softw. Pract. Exper.*, vol. 42, no. 7, pp. 799–816, Jul. 2012.

[4] O. Sander, T. Sandmann, V. Vu-Duy, S. Baehr, F. Bapp, J. Becker, H. U. Michel, D. Kaule, D. Adam, E. Lubbers, J. Hairbucher, A. Richter, C. Herber, and A. Herkersdorf, "Hardware virtualization support for shared resources in mixed-criticality multicore systems," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–6.

[5] M. Weiser, "The computer for the 21st century," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 3, no. 3, pp. 3–11, Jul. 1999.

[6] D. Adam, J. Froeschl, U. Baumgarten, A. Herkersdorf, and H.-G. Herzog, "Cyber Organic System-Model – New Approach for Automotives System Design," in *Electrics/Electronics in Hybrid and Electric Vehicles and Electrical Energy Management (EEHE 2015)*, Apr. 2015.

[7] D. Adam, "Concept for the safe and secure use of android as a runtime environment for apps in the area of automotive infotainment," Master's thesis, Technische Universität München, Oct. 2012.

[8] R. Kaiser and S. Wagner, "Evolution of the PikeOS microkernel," in *First International Workshop on Microkernels for Embedded Systems*, 2007, p. 50.

[9] J. Brygier, R. Fuchsen, and H. Blasum, "PikeOS: Safe and secure virtualization in a separation microkernel," SYSGO, Tech. Rep., 2009.

[10] J. Liedtke, N. Islam, and T. Jaeger, "Preventing denial-of-service attacks on a μ-kernel for weboses," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, 1997, pp. 73–79.

[11] S. Fisher, "multi-core," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, 2013, pp. 73–79.

[12] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[13] SR-IOV, *Single Root I/O Virtualization and Sharing Spec.*, 1st ed., 2007.

[14] *Intel Virtualization Technology for Directed I/O*, Intel, 2011.

[15] V. Vu-Duy, T. Sandmann, S. Baehr, O. Sander, and J. Becker, "Virtualization support for fpga-based coprocessors connected via pci express to an intel multicore platform," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International*. IEEE, May 2014.

[16] ARM, *AMBA AXI and ACE Protocol Specification*, 2012, specification.

[17] Xilinx, *VC709 Evaluation Board for the Virtex-7 FPGA*, 2012, uG.

[18] ——, *Virtex-7 FPGA Gen3 Integrated Block for PCI Express*, 2012.

[19] O. Sander, S. Baehr, E. Luebbers, T. Sandmann, V. Vu-Duy, and J. Becker, "A flexible interface architecture for reconfigurable coprocessors in embedded multicore systems using pcie single-root i/o virtualization," in *The 2014 International Conference on Field-Programmable Technology (ICFPT 2014)*. IEEE, December 2014.

[20] S. Tverdyshev, *PikeOS: Multi-Core RTOS for IMA*, SYSGO AG.

[21] *Guide for using ELinOS on the demo server*, SYSGO AG, September 2012.

[22] *PikeOS Kernel Reference Manual*, SYSGO AG, 2012.

[23] *using PikeOS*, SYSGO AG, 2012.

[24] M. Strasser and P. Sevnic, "A software-based TPM emulator for Linux," *Term Paper, ETH Zurich*, 2004.