VOSYSVirtualNet: Low-latency Inter-world Network Channel for Mixed-Criticality Systems

Julian Vetter*, Jérémy Fanguède*, Kevin Chappuis* and Daniel Raho*
* {j.vetter,j.fanguede,k.chappuis,s.raho}@virtualopensystem.com
Virtual Open Systems, 38000 Grenoble France

***Abstract*—Integrating multiple subsystems with different levels of criticality is a well established concept in the automotive domain. To ensure proper temporal and spatial isolation, a highly privileged software component is installed to orchestrate the subsystems. VOSYSmonitor is such a solution, it enables the co-execution of two operating systems on a single System on Chip - A rich operating system, such as Linux, along with a safety critical operating system, fully isolated from each other using ARM TrustZone. But if we take a closer look at specific automotive scenarios (e.g., "displaying warning signs"), reveals that an interaction of the two operating systems might be desirable.**

In this paper we address this challenge. We present the implementation of a low-latency inter-world network channel. It is built around already existing primitives in both worlds, only implementing the physical layer of the network channel. This ensures a low complexity, meaning only minor modifications have to be made to both operating systems. To prove the feasibility of our design, we built a full prototype that enables a network communication between the two operating systems, while ensuring a proper encapsulation of the safety critical operating system. To validate low reaction times, the design is evaluated with respect to network latency. To complement the measurements, we also performed a number of bandwidth measurements. Finally, we thoroughly discuss potential threat scenarios arising from the network link and how they can be addressed with appropriate countermeasures.

I. Introduction

The Automotive domain is currently undergoing a paradigm shift. Manufacturers alter their system architectures from having a single processing unit for individual subsystems, towards the integration of multiple subsystems with a full spectrum of different safety requirements. In-Vehicle Infotainment systems for example, reside on the very low end of this criticality spectrum and the car control unit resides on the very high end. All are integrated on a single Electronic Control Unit (ECU), by capitalizing on the performance of powerful heterogeneous multi core platforms.

But, these automotive mixed-criticality systems [1] pose new challenges on the system software [2]–[4]. To accommodate for the increased complexity, automotive manufacturers leverage alternative software architectures to execute several software stacks concurrently. In this context, a highly privileged orchestrating entity provides full spatial and temporal isolation of the different software components. Virtualization plays a key role in this trend, but off-the-shelf virtualization solutions (e.g. Xen [5], KVM [6], Hyper-V [7], etc.) are more trimmed towards performance rather than the maximum level of isolation (cf. [8]–[10]). Therefore, they do not provide the level of temporal and spatial isolation required by the automotive domain. Also, with regards to certification (e.g. ISO 26262 [11]) these conventional solutions are lacking behind for the above reasons. Instead, automotive manufacturers use highly specialized certified kernels [12]–[14] to fulfill the demanded safety requirements. These solutions are very similar in their design (e.g., leveraging hardware processor extensions such as ARM TrustZone [15], [16] or ARM VE [17]), but have more predictable timing characteristics, than their off-the-shelf counter parts.

Running on top of these privileged components, the General Purpose Operating System (GPOS) Linux found a widespread adoption due to its versatility and ability to produce a rich user interface for the In-Vehicle Infotainment (IVI) system. It is accompanied by a Special-Purpose Operating System (SPOS), which handles mission-critical tasks (e.g., displaying speed, engine torque and/or warning signs).

In this context, both components serve a dedicated purpose, without the perceived need of interaction. However, a closer look at specific scenarios, reveals that an interaction between systems is required. Then, the strict spatial isolation — initially one of the key requirements of the said system architecture — must be thinned. In this context, establishing a network link between the components raises two questions. First, can the latency requirements from the critical application, be fulfilled? Second, can the integrity of the critical application be upheld, while efficiently exchanging information with the non-critical system?

In this paper these open research questions are addressed. We present the design of an architecture that enables a low-latency network link between SPOS and GPOS, taking safety and security requirements into account. Its feasibility is demonstrated with a full prototype implementation called VOSYSVirtualNet. The prototype is based on a highly privileged firmware called VOSYSmonitor [13], that runs in the monitor mode of modern ARM-based System-on-Chips (SoCs). The communicating endpoints of VOSYSVirtualNet are a Linux running in the non-secure world and a FreeRTOS running in the secure world. We verify the low-latency with a number of benchmarks and put the results into context by comparing them against a reference system. Finally, the security concerns are addressed by referring to the safety and security aspects that are included in the design of

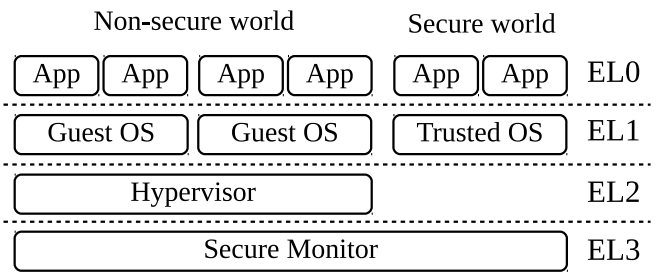

Fig. 1: The hierarchical processor modes of an ARMv8 processor and the orthogonal security concept introduced with ARM TrustZone, that splits the processor modes into two worlds ("secure" and "non-secure").

VOSYSVirtualNet.

The rest of the paper is structured as follows. Section II provides preliminaries and background information. The architecture of VOSYSVirtualNet is presented in Section III and evaluated in Section IV. Then, security implications of our design are discussed in Section V, while related work is presented in Section VI. Finally, we conclude and present future work in Section VII and Section VIII, respectively.

II. Preliminaries & Background

This section describes some background that is crucial for understanding the rest of the paper without requiring the reader to be a priori familiar with these details.

A. ARMv8 Exception Levels

Commonly an ARMv8 application processor features four exception levels. Figure 1 depicts the hierarchical system layout. Whereas, user applications are executed in EL0, the least privileged mode, while EL1 usually hosts the GPOS kernel, e.g. Linux. EL2 is intended to be used by a hypervisor component (e.g. KVM, Xen, etc.). Software executing in this mode is able to install trap mechanisms for certain EL0 and EL1 instructions. Also, a staged paging mechanisms allows software in EL2 to convert the pages translated by EL1 once more. Finally, there is EL3 that usually hosts a firmware to enable the interaction between "non-secure" and "secure world". Further details on this are given in the next section.

B. ARM TrustZone

In 2004 ARM introduced a new security extension called TrustZone [15], [16]. The separation concept operates orthogonal to Exception Levels (ELs) and splits the processor state into two worlds (secure world and non-secure world). Along with the separation concept, ARM also introduced a new processor mode "Secure Monitor Mode", which resides in the highest exception level (EL3). Through the implementation of a new processor instruction, the `smc` (Secure Monitor Call) [18], non-secure components are able to request services from a component running the secure world.
Moreover, the isolation between the worlds is enforced in combination with other cooperating hardware peripherals. A TrustZone compliant memory controller announces the current security state ("secure" or "non-secure"), on every bus transaction, for devices to handle the request accordingly. The Intellectual Property (IP) core TZC-400 [19] enables the configuration of certain ranges of physical memory as secure, preventing non-secure world accesses. Finally, the standard ARM interrupt controller (GIC) supports the classification of interrupt sources into groups, allowing them to be routed to either the secure or non-secure world.

C. VOSYSmonitor

VOSYSmonitor is a highly privileged software component that runs in the Secure Monitor mode (EL3). It enables the native concurrent execution of two operating systems, such as a safety critical RTOS along with a GPOS. The execution of both 32-bit and 64-bit applications is possible and their isolation is ensured by the means of TrustZone.
Since VOSYSmonitor runs in EL3, the Normal world can still opt for a virtualization solution, such as Linux/KVM, which leverages the ARM VE to instantiate multiple non-critical Virtual Machines (VMs). Yet, the RTOS, running in Secure world, is completely isolated from these applications executing in the non-secure world.
Hardware exception mechanisms, such as interrupts, are used in order to ensure an efficient context switching between the two worlds. Additionally, both Operating Systems (OS) can voluntarily give up their execution time by invoking the `smc` instruction. VOSYSmonitor keeps tight control over these exceptions to ensure a proper operation of each world.

III. VOSYSVirtualNet architecture

VOSYSVirtualNet is embedded into an already existing software stack and as such establishes a network link between two components running on top of VOSYSmonitor. An exemplary system architecture is shown in Figure 2. The non-secure world hosts a GPOS (Linux) in order to provide common non-critical automotive applications (e.g., In-Vehicle Infotainment, Vehicle-to-Everything, etc.), whereas the secure-world hosts a SPOS (FreeRTOS), which handles mission critical applications (e.g., an Instrument Cluster).

Before designing VOSYSVirtualNet, we first formulate a number of requirements that must be fulfilled to properly integrate it into the existing software stack. In the following section, we discuss these requirements as well as the design and the implementation of VOSYSVirtualNet.

A. System Requirements

When developing a new component for mission critical applications, such as VOSYSmonitor, it is important to design it in a way that it is minimally invasive since formal evaluation processes (e.g. ISO26262) are lengthy and complex. Therefore, new software components must be integrated with care. In this context, our design decisions for VOSYSVirtualNet are driven by the following aspects:

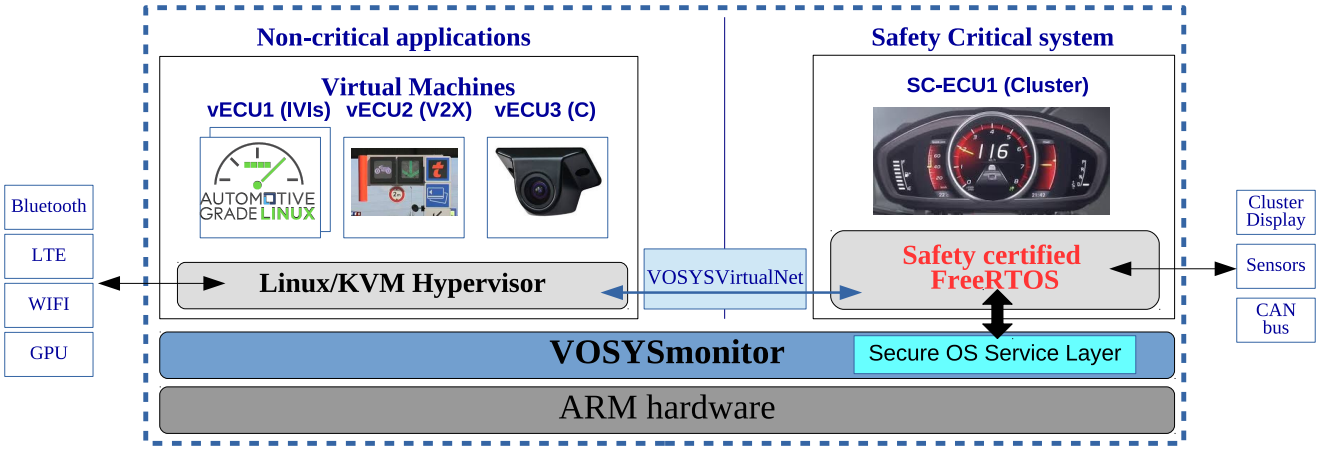

Fig. 2: Exemplary system architecture, with Linux hosting several non-critical applications in the non-secure world, and a safety certified RTOS in the secure world hosting the mission critical instrument cluster. The network link, established by VOSYSVirtualNet, allows to exchange data between both entities.

1) Low latency: It is important to emphasize at this point that the key requirement of our architecture is to achieve a very low latency via the virtual network link. Indeed, the critical OS, running in the secure world, must be able to forward information to the non-secure OS with a very low delay. Thus, all our design decisions are build around this "low latency" requirement.

2) Minimally invasive: Our goal is to make as few changes as possible to all involved software components. Especially, changes to both OSs, running on top of VOSYSmonitor, are problematic, because source level access to the OS kernel is not always guaranteed. But also changes to components where source level access is possible can be problematic. The Linux kernel for example is such a fast-evolving software project and if the modifications can not be applied into the upstream version, an external patch has to be maintained constantly.

But also changes to VOSYSmonitor should be considered with care, because they have to comply to the ISO26262 specification, making the integration of complex code modifications a lengthy process. Thus, our goal is to generate only a minimal patch for VOSYSmonitor and for the Linux kernel to make sure that the porting effort is kept to a minimum (the specific numbers are given in Section III-B).

3) Small hardware requirements: The number of ARM-based SoC is huge and each has different types and amount of hardware peripherals. Therefore, our efforts are focused to only utilize hardware resources that are available on all or at least on a large fraction of ARM-based application processors. Moreover, we only wanted to utilize a small number of these hardware resources.

For the design of VOSYSVirtualNet, a form of signaling mechanism is needed in order to notify the respective world about new network packets. However, external IRQs are arbitrarily assigned by the SoC manufacturers to specific hardware peripherals. Therefore, we decided to use an SGI (Software Generated Interrupt) for the VOSYSVirtualNet signaling mechanism. Each ARM-based SoC equipped with a GICv2 [20] interrupt controller has 16 SGIs (interrupt IDs 0 - 15). Although, the Linux kernel utilizes some of these SGIs (e.g., core synchronization), there are still a number of them left, and it is possible to use one for the signaling of VOSYSVirtualNet.

4) Security & safety awareness: VOSYSVirtualNet is integrated into a safety critical environment, therefore it is important that malicious or erroneous applications in the non-secure world cannot influence critical components in the secure world. To ensure this, we set a rate limit on how frequent one component can signal the respective other component. Also, several security considerations are taken into account (a thorough discussion on this topic is conducted in Section V).

B. VOSYSVirtualNet design

In this section the design of VOSYSVirtualNet is described. It relies on two memory buffers as well as the signaling mechanism. Both aspects will be discussed in the following sections.

1) Memory buffers: The network packets to be exchanged are stored in two shared buffers. The buffers are placed in one of the RAMs available to the system. In general, the placement (allocation) of the buffers is specific to the system software where VOSYSVirtualNet is implemented on. Still, a number of requirements are platform agnostic.

First, due to the limited availability of fast SRAM on the evaluation platform the buffers are allocated in the DRAM. However, it is important to note that the type of RAM might have a positive impact on the performance of VOSYSVirtualNet. Second, since we chose a simple message framing protocol, the buffers have to be fully contiguous. We decided against a scattered design (e.g., based on a linked list) to keep the complexity low. But this means, the OS must provide an API to allocate *larger* chunks of contiguous memory. Finally, before VOSYSVirtualNet is operational, secure and

non-secure world exchange the location of the buffers during a handshaking phase.

2) Signaling: Each entity has access to both buffers, one for the transmission of packets and the other one for the reception. In this context, one entity's "receive buffer" is perceived as the others entity's "transmit buffer" and vice versa. The layout can be obtained from Figure 3. As shown in the figure, each buffer is preceded by a fixed management structure, which besides holding two pointers (the *TX pointer* and the *RX pointer*) also contains a flag to indicate the link status (*up* or *down*). The receive buffer, along with its management structure, is solely used to keep track of new data for reception. The transmit buffer on the other hand is used to send new data.

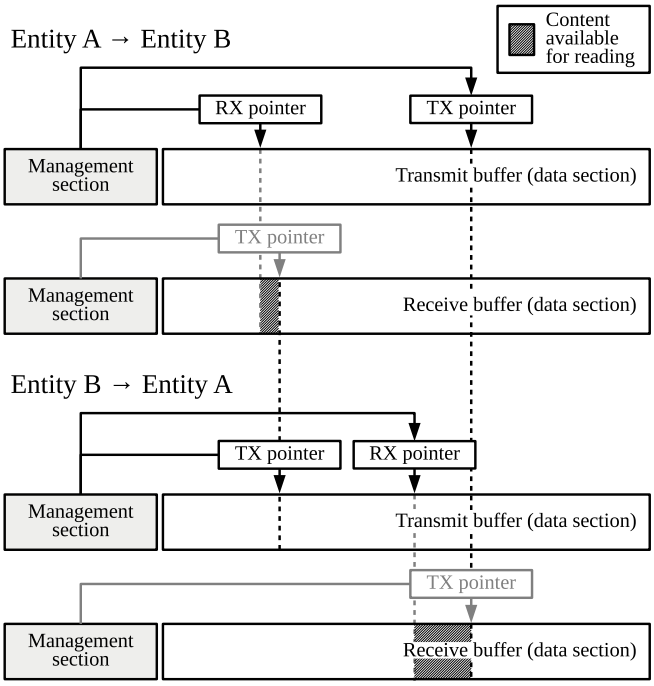

Fig. 3: Layout of the buffers as perceived by both entities.

The RX pointer of each entity's transmit buffer follows the TX pointer of the other entity. If entity A writes new content into its buffer, its TX pointer is adjusted. When entity B is then scheduled, it observes its RX pointer and compares it to the TX pointer in the receive buffer. If they differ, it means new packets are waiting to be handled. In this example, both TX pointers are ahead of the RX pointer, meaning there are new packets available (represented by the hatched space in each receive buffer).

For transmitting data, a simple framing protocol is used. Each packet, which is stored in the buffer by one of the entities, is prefixed with a 32 bit value, holding the size of the following packet. So, when an entity wants to send new data, it first copies the size of the packet to the current position of the TX pointer in the transmit buffer. Then, it increments the TX pointer by 4 Bytes. Afterwards, it copies the network packet into the buffer and increments its TX pointer again by the size of the packet. Depending on the position of the TX pointer, either the driver directly invokes a signal to notify the other world about new content in its buffer (if the TX pointer almost reaches the end of the buffer) or it exits its sending routine (if the TX pointer does not yet exceed the size of the transmit buffer).

The invocation of the transmit signal generates a trap into VOSYSmonitor, which will in turn set the according SGI pending in the receiving world (secure or non-secure). Next time the corresponding world is scheduled, it will immediately trap the SGI into its exception handler in order to manage the incoming interrupt. Then, the Interrupt Service Routine (ISR) schedules a function that takes care of the packet handling. It is important to note that, as described in Section III-A4, one major system requirement is a rate limit for the invocation of the packet handling. By not performing the packet handling directly in the ISR, these operations are fully decoupled.

Once the receiver function is scheduled, it will compare its RX pointer (in the transmit buffer) to the other entity's TX pointer (the one in the receive buffer). If they are different, it means a new content is available to be forwarded to the respective IP stack (see Figure 3). Finally, the receiver will copy the individual packets into the IP stack and increment its RX pointer (in the transmit buffer) accordingly.

C. Implementation

VOSYSVirtualNet is designed in a generic way and the individual implementations decoupled from the architectural design as much as possible. Of course, there are still several design decisions that are unique or specific to a certain software stack and/or OS. Therefore, the implementation peculiarities specific to our chosen platforms are described in this section.

As for the OS, Linux executes in the non-secure world, while FreeRTOS [21] executes in the secure world. Linux brings it's own IP stack, which is used in our implementation. FreeRTOS on the other hand must be extended by an external IP stack. But, under the name FreeRTOS+, the FreeRTOS ecosystem provides a full TCP/IP stack called FreeRTOS+TCP. The FreeRTOS+TCP stack requires the system integrator to implement a number of low level functions that enable the communication with the stack. These tasks purpose are, e.g., putting the packets on the wire and obtaining packets from the wire and forwarding them to the TCP/IP stack.

a) Buffer implementation: The Linux kernel keeps tight control over the physical main memory. So, we decided to let Linux perform the allocation of the buffers. The network communication is preceded with a handshake phase between secure and non-secure world where the device driver in the Linux kernel allocates the buffers through Linux's kernel API (`kmalloc`). Then, the addresses are translated from virtual to physical addresses and forwarded to the secure world entity.

The buffers are allocated with the memory attribute `GFP_DMA` in order to obtain contiguous (and also uncached) memory from the memory allocator. When requesting memory from the kernel allocator with the flag `GFP_DMA`, it forces the

Linux kernel to search for the amount of requested memory in a contiguous block. Depending on the amount of physical memory that is available to the system (our evaluation platform features 1 GBytes of DRAM), the Linux kernel might not be able to serve the request. In our current prototype, a size of 2 MBytes is chosen for each buffer[1]. Depending on the size of the buffers, the network throughput and overall performance of VOSYSVirtualNet increases or decreases.

b) Signaling implementation: As specified in Section III-A, the changes to the Linux kernel should be kept to a minimum. Invoking the receiver function of our driver is the only modification to the Linux kernel we made. We did not modify any other kernel subsystem or the Linux network stack, except for the registration of an unused SGI handler (`kernel/smp.c`), resulting in only 10 Source Lines of Code (SLOC) to the Linux kernel tree. Of course the driver itself consists of more lines of code (292 SLOC), but it is kept off-tree and can be compiled against different Linux kernel versions.

Another requirement is that the packet handling is fully decoupled from the actual reception of the signaling SGI. In an effort to reduce the number of interrupts when receiving a large number of network packets, the new Linux network API, called "New API" (NAPI) provides means to mitigate interrupts for networking devices in the Linux kernel. Therefore, leveraging the NAPI and only calling the function `napi_schedule` from the ISR implies a decoupled packet handling. Linux will then later schedule the registered driver function to handle the packet. In the actual handler function implemented in the Linux driver, the network packets are copied from the shared buffer into a Linux socket buffer. Then, the function `napi_gro_receive` is invoked to pass the packet into the Linux network stack.

On the FreeRTOS side, the network packet handling is also decoupled from the ISR. The ISR wakes up a task by calling `vTaskNotifyGiveFromISR`, which will later be scheduled by the FreeRTOS scheduler to do the actual packet handling. In the handler routine, the available network packets are copied from the shared buffer into a FreeRTOS+TCP specific network buffer descriptor structure. Afterwards, the handler routine invokes the `xSendEventStructToIPTask` function to forward the packets to the IP stack.

IV. Evaluation

To evaluate the feasibility of our approach, we performed a set of latency as well as bandwidth benchmarks on the ARM Juno Development Platform [22]. The latency benchmarks were performed using the `ping` tool. The bandwidth benchmarks were performed using the `iperf` tool (version 2.0.10). The tool `ping` was invoked with the parameter `-c200` (200 ICMP requests). The tool `iperf` was invoked with the parameters `-t400 -i2` (400 seconds test). We sent the ping requests from Linux to FreeRTOS. For the bandwidth tests we used an `iperf` server implementation on FreeRTOS side and connected as a client from Linux. To have a base line for our results, we performed the same benchmarks on a comparable system architecture.

A. Test setup

As a reference systems, we chose a Linux/KVM setup, because it is readily available in the Linux kernel and VMs can easily be deployed using QEMU [23]. We set up three different configurations of Linux/KVM.

1) Linux/KVM: Due to the nature of the safety architecture (the secure OS executes on a single core), all Linux/KVM benchmarks were also performed on a single CPU core. The host Linux was booted with the parameter `maxcpus=2` and both VMs pinned to the secondary CPU ("`taskset -c 1 ...`"), leaving solely the host OS on the primary CPU. The RAM available to both VMs was limited to 512 MBytes ("`mem=512M`"). Also, all power management features were disabled in the Linux kernel (e.g. "`CPU Frequency scaling`" or "`CPU idle PM support`") to make sure no frequency scaling or power management feature would influence our measurements.

As for the network parameters, QEMU version 2.11.0 was used to spawn two VMs with three different network backend configurations. In the host Linux, a layer 2 bridge interface was created and the two tap devices from the guest VMs attached to it, to form the network link. To enable the recent VirtIO network backend "`virtio-net-pci`" [24], Linux kernel version 4.14-rc1 was used for both, the host as well as for the two guest VMs. The kernel features `CONFIG_VIRTIO_NET`, `CONFIG_VIRTIO_PCI`, `CONFIG_VHOST` and `CONFIG_VHOST_NET` were enabled to leverage the backend.

Our Linux/KVM baseline configuration used the QEMU parameter "`-net nic,model=virtio -net tap,...`". To obtain results for a setup with the VirtIO PCI backend the following QEMU parameters were used "`-netdev type=tap,...`" and "`-device virtio-net-pci,...`". Finally, for the VirtIO PCI + VHost setup the first parameter was modified to "`-netdev type=tap,...,vhost=on`".

2) VOSYSVirtualNet: We integrated a prototype into the following software components to evaluate the performance of the VOSYSVirtualNet architecture. In the non-secure world, the network driver was integrated into a Linux kernel (version 4.0.0-rc7). Like, the Linux/KVM setups, we set the parameter `maxcpus=1` to limit the number of CPUs and the parameter `mem=512M` to limit the RAM available to Linux to 512 MBytes. In the secure world, the VOSYSVirtualNet driver was integrated into FreeRTOS v9.0.0 (release 160919). FreeRTOS+TCP (release 160919) was used to provide a network stack in the secure world. We used the heap implementation `heap_4.c` in FreeRTOS with 32 MBytes of memory assigned to the heap. Both buffer allocation strategies in FreeRTOS+TCP

[1]We experimented with different buffer sizes and determined it, to be an appropriate value. The maximum amount of memory that could be obtained from the Linux allocator is 8 Mbytes for each buffer. For bigger allocations, the `kmalloc` call returned `-ENOMEM`.

TABLE I: Latency results for the evaluated architectures (results are in *milliseconds*, lower is better).

Link	Network	Result	Deviation
Linux VM/Linux VM	VirtIO	0.911	1.31200
Linux VM/Linux VM	VirtIO PCI	0.869	1.20400
Linux VM/Linux VM	VirtIO PCI+VHost	0.705	1.42900
Linux/FreeRTOS	VOSYSVirtualNet	0.173	0.00361

(`BufferAllocation_1.c`, `BufferAllocation_2.c`) were explored. The first one allocates a fixed amount of network buffer descriptors during initialization, whereas the other allocates them on demand from the heap. But neither strategy led to major performance gains nor degradations. Thus, `BufferAllocation_1.c` was used with the variable `ipconfigNUM_NETWORK_BUFFER_DESCRIPTORS` set to 10.

B. Benchmarks

In this section, we present the results from different latency and bandwidth benchmarks. Each measurement was performed on the previously discussed hardware/software setups.

The Linux/KVM configurations show overall average latency results. For the general use-cases that are covered by the virtual network link between two VMs these numbers are sufficient. But taking a closer look, reveals that the results show a very high deviation, which is not acceptable for most "high-criticality" use-cases. The detailed latency results

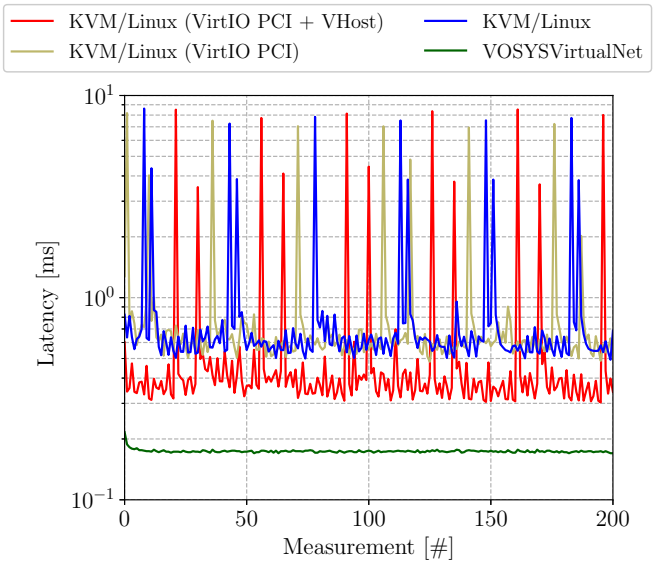

Fig. 4: Latency results.

(depicted in Figure 4), reveal that the Linux/KVM setups have highly fluctuating packet arrival times. These spikes of greatly increased latencies in regular intervals lead to the relatively high average and also a high standard deviation (shown in Table I).

The matter itself was not investigated any further, because the increased latency interval pattern is different for every Linux/KVM setup. The issue seems to be due to peculiarities in the VirtIO implementations. With VOSYSVirtualNet, we achieved a more consistent and also ∼2-3x lower latency. This aspect is especially critical for a system built for the automotive domain, since it handles mission critical data in the SPOS, where low response times are crucial.

A comparison between the throughput results is depicted in Figure 5. Here, the Linux/KVM setup clearly shows it's strength, with throughput values of up to ∼1.9Gbps with VirtIO PCI and VHost enabled. Whereas VOSYSVirtualNet achieves a bandwidth of ∼38Mbps in TCP mode. The reduced

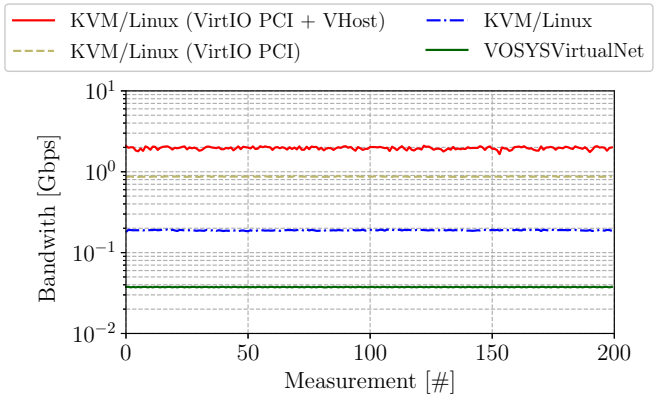

Fig. 5: Throughput results.

bandwidth in VOSYSVirtualNet is due to the scheduling policy that is enforced in VOSYSmonitor. Whenever the SPOS has work to perform, interrupts are masked for the GPOS, not allowing it to send further packets. This is of course intended behaviour and part of the security measures, that is enforced on the network channel (and on the non-secure world in general) by VOSYSmonitor.

Moreover, Linux/KVM with any VirtIO backends uses Large Receive Offload (LRO) in the Linux kernel. This feature enables the aggregation of multiple incoming packets from a single stream into a larger buffer before they are passed higher up the networking stack. This reduces the number of packets that have to be processed. To improve the overall performance of VOSYSVirtualNet with respect to throughput we adapted the Maximum Transmission Unit (MTU) size. We could perform this optimization because we fully control the virtual network link, without the packets reaching a physical link. When the packets are forwarded onto a real network link, the Linux network stack takes care of cutting the network packets once again.

When using a MTU size of 1500 Bytes (default in Linux) the throughput is as low as ∼10 Mbps. An increased MTU size of up to 65 kBytes leads to four times better throughput results of up to ∼40 Mbps. However, the throughput gain does not scale linearily, because bigger packets, take considerably more handling time in the receiving world. The results can be obtained from Figure 6. When increasing the MTU the time the system spends in the receiving entity to handle the packet increases. The error bars however indicate that even with a

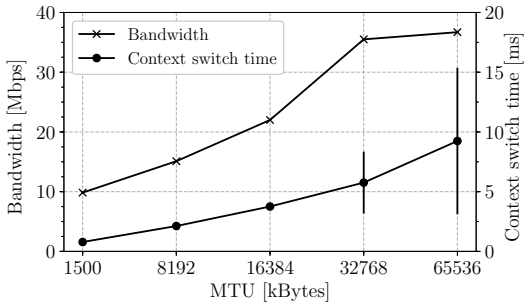

Fig. 6: MTU size and its impact on the achievable throughput of VOSYSVirtualNet.

packet size of up to 65536 the time for the packet handling can be as low as ∼2.7 ms but on some occasions can go up to ∼15 ms.

V. Security Implications & Discussion

A communication between non-secure and secure world has security implications. In this section, the most critical ones are discussed, and how VOSYSVirtualNet can deal with them.
Although complex functionality (e.g., packet framing, signaling, etc.) in the non-secure world is contained in a system driver and therefore sealed away from malicious user processes, an adversary, which manages to take over the non-secure OS layer, can perform a number of attacks, and undermine the spatial and/or temporal isolation enforced by VOSYSmonitor.
Of course, it is important to note that the highlighted threats can in the same way arise from a malicious or malfunctioning component in the secure world.

A. Denial-of-Service

In general, a Denial-of-Service (DoS) attack describes a scenario, where an adversary sends requests to a remote endpoint at such a high rate that the receiver is only able to serve the adversaries requests, without the ability to perform any other task [25], [26]. In a worst case scenario, the receiver completely shuts down due to, e.g., memory scarcity [27] or CPU starvation [28].
A similar scenario can arise when two entities communicate over a virtual network link (such as VOSYSVirtualNet). Indeed, if one entity sends packets at a high rate, the receiving entity might entirely be occupied by handling those requests. Even though, the secure world has a higher priority than the non-secure world, the non-secure world still might be able to DoS the secure world. This is highly dependant on how the priorities are set in the secure world. If, e.g., the IP stack runs with a very high priority in the secure world the non-secure world could fill the entire shared buffer with packets, then the secure-world might only be occupied by processing these packets without being able to perform any other task.

Countermeasure: To address this issue in VOSYSVirtualNet neither of the two entities (secure and non-secure) performs the packet processing directly in their respective ISR. Instead, the ISR adds a task to the respective scheduling queue (in Linux terminology called "tasklet"), which is later dispatched by the systems regular scheduling policy to perform the actual packet processing. The handling task has a lower priority and can therefore be interrupted by critical processes on each side. To decrease the severity of DoS attacks in the critical OS even further, the priority of the task that contains the IP stack can also be reduced. But, the selected priorities must be adapted towards the characteristics of the system, the number of concurrent tasks, and the importance of the virtual network link.

Another counter measure is to limit the number of packets that are processed during a scheduling period in the receiving entity. This can be achieved by reducing the size of the "transmit buffer" to an appropriate size or by introducing an artificial value that interrupts the processing after n packets. The Linux kernel's NAPI already provides such a mechanism called $budget$. Per invocation of `napi_schedule` a maximum of $budget$ packets are processed. The NAPI documentation suggests a value of 64 for faster interfaces and a default of 16. Currently, VOSYSVirtualNet uses a conservative $budget$ value of 16, to limit the severity of DoS attacks, in favor over raw performance. In FreeRTOS we implemented the same mechanism limiting the number of packets that are processed per invocation of the handler task.
Even though currently not implemented by VOSYSVirtualNet is a rate limiter directly embedded into VOSYSmonitor. VOSYSmonitor would count the number of incoming `smc` calls from either entity and dismiss them if they exceed a certain number.

B. Packet corruption

Data parser components have to deal with unsanitized input making them vulnerable to attacks [29], [30]. An IP stack is no exception, parsing corrupted network packets (purposely or unpurposely, due to, e.g., a corrupted user application) is challenging and might lead to unpredictable behaviour at the receiving entity. Depending on the robustness of the packet parser of the IP stack, the attack might have a severe impact on the availability of system software on the receiving side. The attacks severity is further influenced by two aspects. First, does the IP stack run with system privileges (as part of the OS kernel) or as a user application that can be shutdown? Second, is the IP stack a component in a vital part of the secure worlds functionality or is it an extra component that in case of failure can be shut down?

Countermeasure: Invoking the IP stack with corrupted network packets is a common attack scenario and the IP stacks on both sides should be robust enough to handle such scenarios. However, several CVEs show that the Linux kernel suffered from such an issue [31], [32] in the past. The same vulnerability applies to the secure OS but the issue highly depends on the used IP stack as well as on the used secure OS.
Generally, it is advisable to run complex components, such as an IP stack, in an isolated user application and not with

system privileges (even though this means a diminishing return in terms of performance). FreeRTOS already follows the microkernel [33] approach, running system components in user tasks. Meaning, the FreeRTOS IP stack runs in a user application, and thus limiting the impact of a crash or takeover by an adversary.

C. Memory corruption

The non-secure component is responsible to allocate the buffers as well as to populate their addresses to the secure world. Granted that a malicious entity in the non-secure world allocates the shared buffers just at the boundary to the secure world (storing the management structure still in the non-secure part of the memory), leaving the signaling intact but provoking the secure world to overwrite parts of its own memory when storing a network packet in the buffer.

Countermeasure: The countermeasures to overcome this issue are however straight forward. By checking the location of the secure memory and comparing it to the buffer locations retrieved from the non-secure world, the secure world can make sure to not disclose any security critical information, or worse overwrite its own memory.
In the worst case, the virtual network link could not be established, but security of the critical OS is not jeopardized.

VI. Related Work

In the following section we present a number of topics, which are relevant to this work. While, there has not been a lot of research in the domain of inter-world (secure/non-secure) communication, our architecture closely resembles a virtualization architecture. Therefore, we focus our related work towards research that optimizes a virtual link between VMs (inter-VM communication).

Early work on the subject has been done by Wang et. al. [34] in 2008. To improve the inter-VM communication speed in Xen, they implemented XenLoop and achieved bandwidths of $\sim$4000 Mbps and latencies of $\sim$28 microseconds between two VMs on a x86-based workstation (dual-core Intel Pentium D 2.8 GHz CPU, 4 GBytes of RAM).

In 2009, Burtsev et al. introduced Fido [35] an improved inter-VM communication mechanism. The novelty of their approach was to leverage the relaxed trust model between two software components in an enterprise appliance to achieve higher throughput rates. Indeed, their evaluation results suggest high bandwidths by doubling the bandwidth result that was achieved by Wang et al. with XenLoop. Their benchmarks were performed on an high-end x86-based workstation (two quad-core AMD Opteron 2.1 GHz CPUs, 16 GBytes of RAM) and they achieved bandwidth results as high as $\sim$10000 Mbps with a message size of 64 KBytes and latency results between $\sim$30 to $\sim$90 microseconds with different packets sizes.

Ren et al. [36] did an extensive study on the current state of the art on Inter-VM communication. Their survey compares the latest solution for the two most common open source hypervisors Xen and Linux/KVM. Their measurements are performed on a number of different hardware platforms, ranging from a workstation grade machine with a 2.8 GHz CPU, 4 GBytes of RAM up to, a server grade machine with two quad core CPUs with 2.67 GHz and 48 GBytes of RAM. On Linux/KVM, they achieved bandwidths of $\sim$6000 Mbps with a solution called MemPipe [37] and $\sim$8000 Mbps with XenLoop. Their outcomes are in line with the previous findings and results from Burtsev et al. and Wang et al. They however, do not provide absolute latency numbers, only setting the different solution in a relative context.

VII. Conclusions

This paper raised the research question, whether a low-latency virtual network link can be established between the secure and non-secure world, connecting two software components with different levels of criticality, while still being resilient against manipulations from one of the entities. We addressed the question by introducing VOSYSVirtualNet, a virtual network link that enables communication between a GPOS in the non-secure world and a SPOS in the secure world. The solution is integrated into an existing software stack that is based on a highly privileged component called VOSYSmonitor, executing in EL3, which orchestrates the communication process and ensures a fair assignment of time to each entity. With a full prototype implementation of VOSYSVirtualNet and a number of latency benchmarks we proved its feasibility and verified our design. To make it resilient against attacks we integrated a number of countermeasures into our design in order to ensure a proper spatial and temporal isolation, while the virtual network link is up.

VIII. Future Work

A network link established with VOSYSVirtualNet provides a low latency, which is the main goal of our architecture. However, VOSYSVirtualNet can be extended by a number of aspects in the future.

For the demonstrator, we choose FreeRTOS as a SPOS running in the secure world. But, the integration of our design into a fully AUTOSAR [38] compliant SPOS still stands out.

Also, the use-cases that were driving our design decisions had low-latency requirements, whereas the bandwidth requirements were neglectable. Currently, our prototype of VOSYSVirtualNet achieves a bandwidth of $\sim$38Mbps (measured with `IPerf`). But, it is important to note, that the implementation in the secure world is very OS specific, and has to adhere to the scheduling peculiarities of FreeRTOS. Tests with increased MTU sizes already suggest that higher bandwidth results can be achieved in the current setting. Also utilizing the LRO, which is already present in VirtIO is future work.

Acknowledgments

This work was supported by the dRedBoX project. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 687632. This work reflects only the authors' view and the European Commission is not responsible for any use that may be made of the information it contains.

REFERENCES

[1] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, pp. 1–69, 2013.

[2] D. De Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*. IEEE, 2009, pp. 291–300.

[3] S. Trujillo, A. Crespo, and A. Alonso, "Multipartes: Multicore virtualization for mixed-criticality systems," in *Digital System Design (DSD), 2013 Euromicro Conference on*. IEEE, 2013, pp. 260–265.

[4] R. Ernst and M. Di Natale, "Mixed criticality systems—a history of misconceptions?" *IEEE Design & Test*, vol. 33, no. 5, pp. 65–74, 2016.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 164–177.

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux Virtual Machine Monitor," in *Proceedings of the Linux symposium*, vol. 1, 2007, pp. 225–230.

[7] A. Velte and T. Velte, *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[8] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory Deduplication as a Threat to the Guest OS," in *Proceedings of the Fourth European Workshop on System Security*. ACM, 2011, p. 1.

[9] J. Rutkowska and A. Tereshkin, "Bluepilling the xen hypervisor," *Black Hat USA*, p. 27, 2008.

[10] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.

[11] ISO, "Road vehicles – Functional safety," 2011.

[12] Y. Li, R. West, and E. Missimer, "A virtualized separation kernel for mixed criticality systems," in *ACM SIGPLAN Notices*, vol. 49, no. 7. ACM, 2014, pp. 201–212.

[13] P. Lucas, K. Chappuis, M. Paolino, N. Dagieu, and D. Raho, "VOSYS-monitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A," in *29th Euromicro Conference on Real-Time Systems*, 2017.

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

[15] "ARM Security Technology - Building a Secure System using TrustZone Technology," Whitepaper, ARM Limited, April 2009.

[16] Alves.T and Felton.D, "Trustzone: Integrated hardware and software security-enabling trusted computing in embedded systems," Whitepaper, ARM Limited, July 2004.

[17] R. Mijat and A. Nightingale, "Virtualization is coming to a platform near you," *ARM White Paper*, 2011.

[18] "SMC CALLING CONVENTION System Software on ARM Platforms," Whitepaper, ARM Limited, November 2016.

[19] "ARM CoreLink TZC-400 TrustZone Address Space Controller," Technical Reference Manual, ARM Limited, September 2015.

[28] W. O. Chee, P. Chandra, J. Williams, S. Jansen, and R. Barnett, "Checkmate with denial of service," Black Hat Briefings, September 2011.

[20] "ARM Generic Interrupt Controller. Architecture version 2.0," Whitepaper, ARM Limited, July 2013.

[21] R. Barry *et al.*, "FreeRTOS," *Internet, Oct*, 2008.

[22] "Juno ARM Development Platform SoC," Technical Reference Manual, ARM Limited, September 2014.

[23] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[24] W. Wang, J. Nakajima, M. Ergin, J. Tsai, G. Xiao, M. Koujalagi, H. Xie, and Y. Liu, "Design of Vhost-pci," Intel Corp., 2016.

[25] A. D. Wood and J. A. Stankovic, "Denial of Service in Sensor Networks," *computer*, vol. 35, no. 10, pp. 54–62, 2002.

[26] D. Senie and P. Ferguson, "Network Ingress filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing," *Network*, 1998.

[27] W. M. Eddy, "TCP SYN Flooding Attacks and Common Mitigations," 2007.

[29] B. Michéle and A. Karpow, "Watch and be Watched: Compromising All Smart TV Generations," in *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*. IEEE, 2014, pp. 351–356.

[30] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid, "Reins to the Cloud: Compromising Cloud Systems via the Data Plane," *arXiv preprint arXiv:1610.08717*, 2016.

[31] "CVE-2017-1000251." Available from MITRE, CVE-ID CVE-2015-1000251., Dec. 9 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000251

[32] "CVE-2016-10229." Available from MITRE, CVE-ID CVE-2016-10229., Dec. 30 2015. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10229

[33] J. Liedtke, "On μ-Kernel Construction," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 237–250. [Online]. Available: http://doi.acm.org/10.1145/224056.224075

[34] J. Wang, K.-L. Wright, and K. Gopalan, "Xenloop: a transparent high performance inter-vm network loopback," in *Proceedings of the 17th international symposium on High performance distributed computing*. ACM, 2008, pp. 109–118.

[35] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson, "Fido: Fast inter-virtual-machine communication for enterprise appliances," in *In USENIX ATC*. Citeseer, 2009.

[36] Y. Ren, L. Liu, Q. Zhang, Q. Wu, J. Guan, J. Kong, H. Dai, and L. Shao, "Shared-memory optimizations for inter-virtual-machine communication," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, p. 49, 2016.

[37] Q. Zhang and L. Liu, "Workload adaptive shared memory management for high performance network i/o in virtualized cloud," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3480–3494, 2016.

[38] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, "AUTOSAR–A Worldwide Standard is on the Road," in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, vol. 62, 2009, p. 5.