**2023 NDIA MICHIGAN CHAPTER
GROUND VEHICLE SYSTEMS ENGINEERING
AND TECHNOLOGY SYMPOSIUM
MOSA TECHNICAL SESSION
AUGUST 15-17, 2023 - NOVI, MICHIGAN**

Deterministic & Modular Architecture for Embedded Vehicle Systems

Michael Doran[1], Mark Russell[2], Leonard Elliott[2]

[1]DornerWorks Ltd, Grand Rapids, MI
[2]U.S. Army DEVCOM GVSC, Warren, MI

The next generation of Army ground vehicle systems aim to provide the warfighter with advanced capabilities while ensuring cyber resiliency. One key technology, Ethernet, has enabled the modernization of military ground vehicles by providing a broad range of beneficial features. The scalability and high bandwidth of an Ethernet based system provides the ability to process large volumes of sensor data with low latency, however its inherent lack of determinism represents a significant disadvantage. A deterministic network requires that communication assurance is provided through bounded message latency, and this is required for many ground vehicle weapon and crew stations functions. Traditional Ethernet based networks are unable to satisfy the strict safety and functional requirements for Army vehicle systems due to this lack of determinism. Modular Open System Approach (MOSA) initiatives such as the Ground Combat System Common Infrastructure Architecture (GCIA) seek to leverage open-standards such as Time-Sensitive Networking (TSN) to achieve real-time, deterministic communication over Ethernet. TSN provides enhancements to regular Ethernet which enable logical segmentation of deterministic and traditional best-effort network traffic while simultaneously be transmitted on the same physical media.

This paper presents a reference architecture which incorporates key elements from GCIA, including TSN, and complements them with embedded virtualization technologies to enhance the safety and security of the system. The seL4 microkernel is used to deploy virtualized guests and containers on a target representative of an embedded platform for ground vehicle electronics, the ARMv8. By utilizing seL4 and virtualized guests a system designer can now combine the isolation provided by hypervisors with the logical segmentation provided by TSN *to create a partitioned architecture that increases system assurance. Aspects of this architectural approach and technology have already been adapted across multiple programs within DEVCOM-GVSC.*

Citation: Michael Doran, Mark Russell, Leonard Elliott, "Deterministic & Modular Architecture for Embedded Vehicle Systems" In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 16-18, 2023.

1. INTRODUCTION

1.1 Background

In the last decade a key enabler for next generation automotive and industrial system hardware/software designs has been virtualization [1]. Modern embedded systems that utilize the ARM, x86, or RISC-V architectures are now equipped with virtual extensions. These extensions were the response to market pressures to minimize size, weight, power, and cost (SWaP-C).

Software designers are now able to:

1. Architect a system capable of enforcing isolation of disparate compute domains while executing on the same hardware [2]
2. Provide a system capable of rapidly scaling and updating with an existing DevSecOps pipeline [3]
3. Adapt to new industry best practices by deploying improved capabilities (i.e. TSN)

One response to these demands in recent years is hypervisor-based technology, however traditional enterprise hypervisors do not provide the assurance or real-time performance needed to support mixed-criticality applications. The seL4 microkernel can serve as an embedded hypervisor that can support the deployment of high assurance software through its unique and comprehensive formal verification [4].

1.2 seL4

Formal proofs of correctness make seL4 a strong candidate for building safety-critical and secure systems. Figure 1 illustrates the seL4 proof chain.

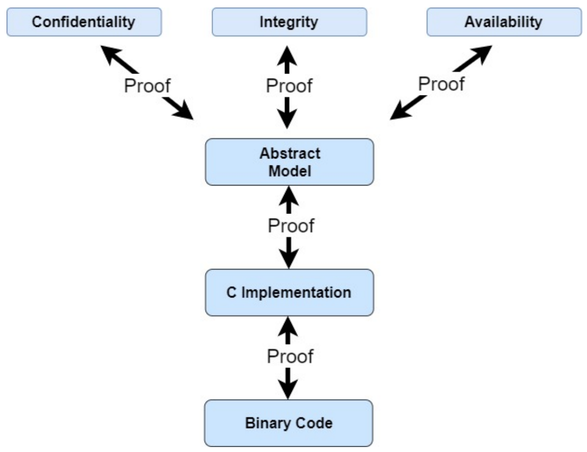


Figure 1: seL4 Proof Chain

The core of seL4's verification is the functional correctness proof, which claims that the C implementation is free of implementation defects. The formal specification of the kernel's functionality is expressed in a mathematical language called higher order logic (HOL). The HOL specification in this case is represented by the abstract model in Figure 1. The C implementation is then a refinement of the abstract model, meaning that the possible behaviors of the C code are a subset of those allowed by the abstract model. Kernel behavior is expressed by the abstract specification, thus preventing the kernel from behaving in ways that are not allowed by the specification. A kernel that is formally verified, such as seL4, can then shield itself from attacks such as stack smashing, null-pointer dereference, and any code injection or control flow-hijacking [1].

As a part of the formally verified proof, seL4 also provides a way to additionally verify the executable binary produced by the compiler. Verifying the executable binary is an additional security-critical step to prevent malicious compilers from performing such actions such as building a Trojan that opens a back door to the OS. Specifically, the binary is proved to be a correct translation of the proved correct C code.

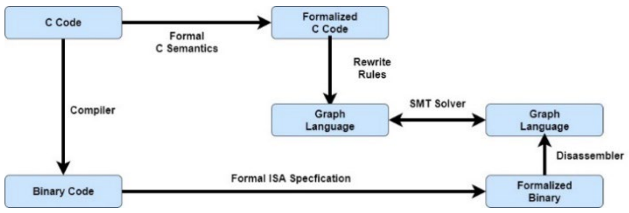


Figure 2: Translation Validation Proof Chain

The proof chain illustrated in Figure 2 is an automatic process that happens in multiple stages. A formal model of the processor's Instruction Set Architecture (ISA) formalizes the binary in the theorem prover. The formalized ISA feeds the disassembler, written in the HOL4 theorem prover, to translate the low-level representation into a higher-level representation in a graph language that represents control flow. The formalized C program is then translated into the same graph language which allows for comparison of two programs to assess for equivalent representation.

1.3 Security Properties

Figure 1 illustrates the proofs between the abstract specification and the high-level security properties: confidentiality, integrity, and availability. These properties are subsets of the abstract specification and build in security to the kernel. When proven correctly the kernel will enforce these properties:

- Confidentiality: seL4 prevents unauthorized read/writes to data.
- Integrity: seL4 prevents unauthorized modification of data.
- Availability: seL4 prevents unauthorized use of resources.

1.4 Hypervisor Design

seL4 provides the ability for a system architect to implement a hypervisor and virtual machine monitors (VMMs) capable of deploying isolated virtual machines (VMs) as notionally illustrated in Figure 3.

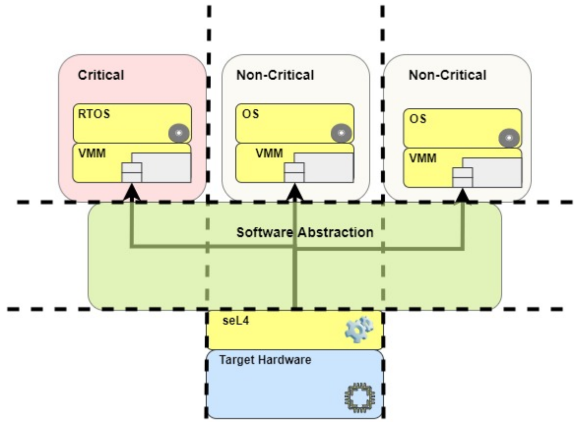


Figure 3: seL4 Hypervisor Mode

The design paradigm depicted in Figure 3 demonstrates a high-level overview of the type of hardware and software configuration designed to meet SWAP-C requirements. At the lowest level of this architecture is the target hardware. In this case the target hardware used for this effort was the ARMv8, however, it can be applied to x86 and RISC-V as well. Software abstraction is applied to achieve a Type 1 hypervisor model. This type of software abstraction enables logical partitioning of compute domains in the form of guest VMs.

1.5 TSN

One of the main objectives of this effort was to achieve determinism within an seL4-based virtualized embedded architecture. Figure 3 illustrated the foundational concept of the embedded design and Figure 4 illustrates the hardware resources needed and the interface that successfully implemented TSN for this effort.

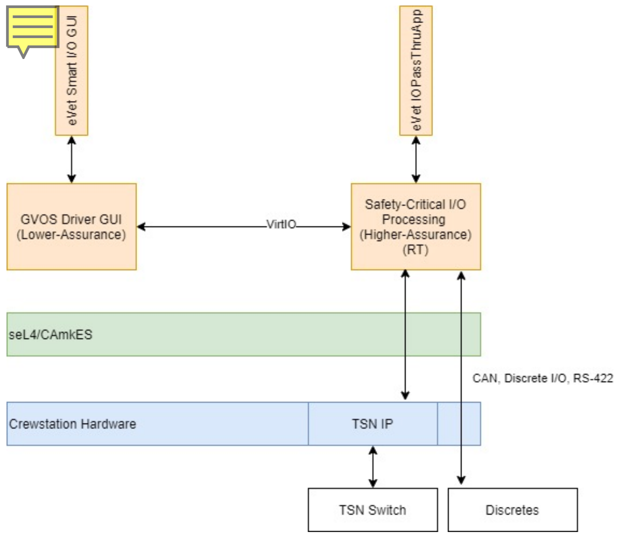

eVet Smart I/O GUI
eVet IOPassThruApp
GVOS Driver GUI (Lower-Assurance)
VirtIO
Safety-Critical I/O Processing (Higher-Assurance) (RT)
seL4/CAmkES
CAN, Discrete I/O, RS-422
Crewstation Hardware
TSN IP
TSN Switch
Discretes

Figure 4: Implementing TSN Bridge with seL4 via Device Passthrough

The architecture in Figure 4 showcases how device interfacing works with seL4/CAmkES. The crew-station hardware in this design is equipped with TSN Intellectual Property (IP) core, discrete I/O, and Ethernet. TSN traffic is routed via a TSN bridge. Once TSN traffic has entered the system the TSN IP core within the crew-station hardware forwards the traffic along to both compute domains via a virtIO connection. The main benefit of this architecture is that the high assurance domain has direct, low latency, access to hardware peripherals, and other domains still have mediated access to those same peripherals via virtIO. This configuration provides flexibility to add/remove hardware peripherals as needed and lends itself well to integration of a ground vehicle system.

1.6 Overview

The rest of this paper will provide context surrounding specific Army uses cases and objectives for this effort. In addition, this paper also provides a detailed overview of the architecture specific features such as TSN Bridging, Virtual CAN, Virtual GPIO, Health Monitoring, and Secure Boot. This is followed up with brief testing results and followed by a discussion regarding transition results and future improvements.

2. Army Use Cases and Objectives

This section aims to provide an overview of the objectives that motivated this effort. Brief details regarding the GCIA design are discussed, which is then followed up with the objectives and Army use cases for ground vehicles.

2.1 Objectives

The objective of this effort was to advance cyber-physical systems and capabilities. This was achieved in this effort by providing a reference architecture that is aligned to ground vehicle platforms, therefore, ensuring consistent MOSA implementations.

The modular design, supported with strong isolation provided by embedded hypervisor and virtual network technology, will reduce time and cost to add or upgrade capabilities on and across future platforms and enable development and integration of solutions at the hardware/software component level versus the entire subsystem. Overall, this design principle should reduce the need to retest the entire system each time a new component is added or changed.

The current state of combat ground vehicle design is undergoing a paradigm shift. Capabilities in the form of sensors, processors, and effectors are now designed to share hardware and compute resources all while maintaining a continuous update/upgrade cycle. Figure 5 illustrates major elements of the GCIA including the relevant network, I/O Adapter, Crewstation, and Common Compute components.

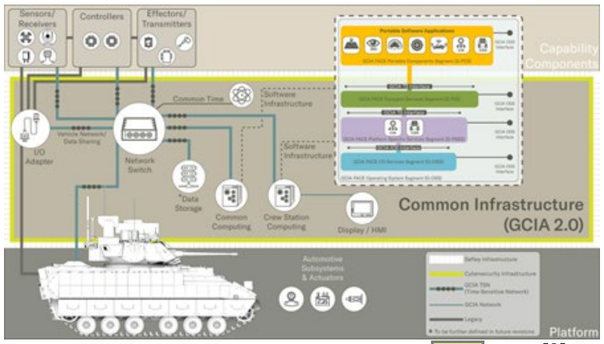


Figure 5: GCIA 2.0 Conceptual Diagram[8]

The scope of effort in this research enables GCIA, as the SmartIO platform can be used as a Crewstation, Common Compute or I/O Adapter and is TSN-enabled. SmartIO converts discrete I/O signals to TSN messages to be distributed multiple end nodes. The end nodes in this case include both the virtualized guests executing in disparate compute domains – example given in Figure 3 and other TSN end stations connected to the TSN bridge. The rest of this section will discuss the specific Army use cases for this type of reference architecture.

2.2 Army Use Cases

The military ground vehicle use-cases and scenarios where this type of modern, partitioned, and deterministic architecture are numerous and support the kinds of logical segmentation identified as a key objective in the DoD's Zero Trust Strategy that was released in November 2022. Traditionally military ground vehicle control applications and functions are implemented with technologies such as CAN, MIL-STD-1553, RS-422 and/or point-to-point discrete signals. These interfaces are all prime candidates to be integrated with and eventually replaced with this emerging architecture. Examples of systems that have enhanced in specific use cases are discussed in this section.

2.2.1 Virtualized Sensor Control for Conformant and Distributed I/O

A ground vehicle system is entirely made up of complex subsystems. The reference architecture developed in this effort utilized a combination of virtualized I/O interfaces and TSN to enable the common compute i.e., SmartIO, to process, transmit, and receive sensor data distributed throughout the vehicle.

Distributed processing and I/O in this context mean that the system must have the ability to route hardware peripheral data, such as CAN, throughout the entire system.

Another benefit of conforming I/O with virtual interfaces is the reusability and portability of that software across different computing architectures (i.e. x86, RISC-V, etc.). More details are provided in Section 6 Transition with respect to the success of this architecture being deployed across multiple programs at DEVCOM-GVSC.

Virtualized conformant I/O also enables security enhancements for a more hardened posture. seL4 provides great access control of hardware components within VM(s) to prevent any undesired access from one subsystem to another. Finally, virtualized conformant I/O provides a lower attack surface by virtue of SWAP-C requirements. With one common compute platform the lower physical footprint results in a smaller attack surface for an Intrusion Detection System (IDS) to monitor.

2.2.2 Health Monitoring and Remote Control

Establishing conformant I/O within a complex system along with a common compute provides the foundation for implementing health monitoring, and remote-control functionality. Custom components provide the ability for a designer to develop a monitor component that can glean info on the internal health of the system. This health monitor component can be configured to continuously monitor for certain system events including, but not limited to:

- Internal faults (crashes, overflows, malfunction, etc.)
- Memory Access read/write)
- Peripheral Access

Internal events and health metrics are easily transmittable to an end node within the same network and can provide real-time information on the current state of the vehicle. Of course, if data can be transmitted from the system it is plausible to design an additional component service to seL4 that allows for incoming commands to be received and parsed throughout the system for remote control. Remote control of a ground vehicle system promotes teleoperation by remote operators.

2.2.3 Secure Boot and Update

Securely booting and updating is of paramount importance for a cyber physical system, especially a weapon system. The reference architecture employed in this effort devised a portable mechanism for securely booting the seL4 microkernel, components, and guest VMs. The architecture is modular enough to provide a consistent secure boot and update process across multiple platforms.

Of course, once a system has deployed a version of software it must adhere to a consistent update process to update elements of that same system. This reference architecture can update individual software components of the system ranging from firmware, microkernel, component, virtualized IO, guest VMs, etc. These integrity protections enhance the overall security posture of the systems.

3. Architecture

This section aims to provide a detailed overview of the reference architecture deployed and evaluated in this study. The UEI SmartIO platform (Figure 6) which is based on the AMD Zynq architecture was the target platform for this effort.

3.1 Hardware

The UEI SmartIO system is a ruggedized rack mounted design. The rugged and compact design expands system capability by installing hardware peripherals to the rack backplane as needed. The Zynq single board computer automatically enumerates and interfaces with additional hardware peripherals. A middleware layer is also provided to make hardware peripherals available through a network API.

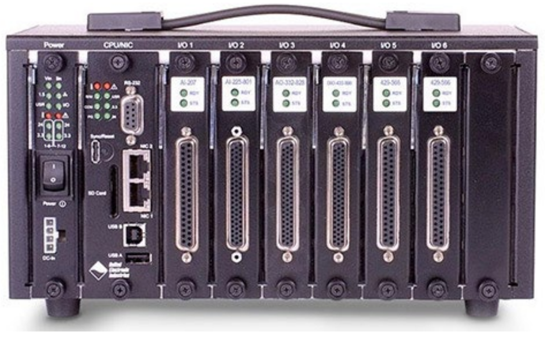


Figure 6: UEI SmartIO Platform

The Zynq platform is a capable architecture that provides processing system (PS) and programable logic (PL) in one integrated circuit. The PS in this case executes the virtualized architecture (seL4, VMM, VMs, etc.) and the PL is dedicated to executing the UEI support functionality including the TSN IP core. Figure 7 provides an abstract view of how the processing system (PS) interfaces with the programmable logic (PL) for the Zynq platform.

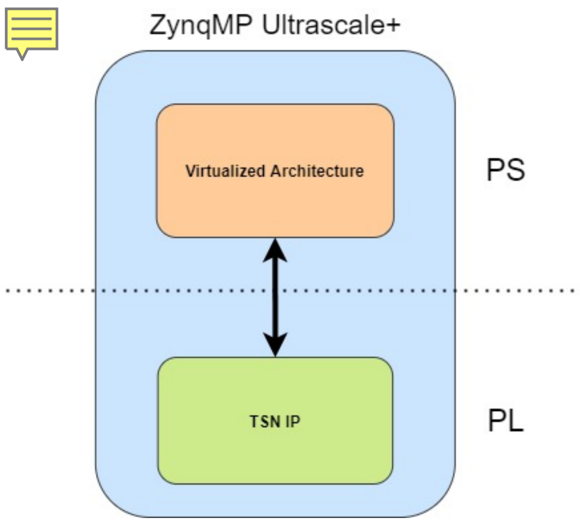


Figure 7: PS-PL Separation of Virtualized seL4 architecture and TSN IP

The UEI SmartIO platform combined with the Zynq architecture is a representative candidate for prototype of next generation technology for ground combat vehicles. The form factor aligns with modular hardware design that allows for easy integration and development of hardware peripherals. The Zynq system on chip (SoC) provides the capability to design for SWAP-C requirements.

Referring to the system architecture drawing in Figure 1, it is easy to think of a mil-ruggedized version of SmartIO as a Crewstation, Common Compute, or I/O Adapter component. The SmartIO interfaces with vehicle capability components, storage, networking, etc.

3.2 Software

The main software elements that were developed because of this effort include:

- Shared Device Support
  - GPIO
  - NVMe
  - CAN
  - Network
- TSN Bridging
- Formally Verified Network Stack
- Fault Tolerance/Handling
- seL4 Application Secure Boot

The rest of this section will discuss the design details of each of the elements listed above.

3.3 Shared Device Support

One of the goals of this project was to provide documented and reusable software modules that can be ported to various hardware platforms. Various hardware platforms can take the form of a different architecture all together (x86, ARM, RISC-V, etc.) or it can mean a different platform within the same family of architecture.

For example, this effort utilized a total of four development kits to develop the software capabilities discussed in this section. The four development kits used were all based on the Zynq. All software was eventually tested on the SmartIO platform for verification. If anything, this testimonial serves to substantiate the claim that this reference design is successful with respect to reusability and porting across platforms.

The rest of this section aims to discuss the software building blocks of each component as it relates to GPIO, NVMe, CAN, and Ethernet.

3.3.1 GPIO

The GPIO seL4 driver is split into ~~two~~ main components: the GPIO-MUX-Server front end and the low-level device driver backend. The low-level device driver is responsible for reading and writing the GPIO device registers, and the GPIO-MUX-Server component is an abstraction that provides a procedural interface to any client connected to it. Figure 8 illustrates the seL4 GPIO driver.

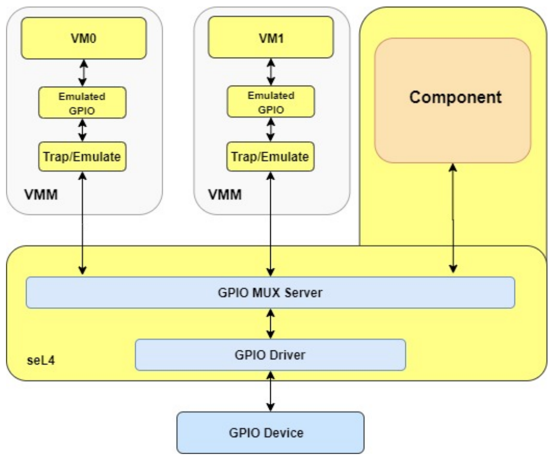


Figure 8: Virtual GPIO Driver

3.3.2 2 NVMe

NVMe driver support enables the capability for this reference design to interface with non-volatile storage. Figure 9 below illustrates the block diagram for how the virtual NVMe driver was implemented for this design.

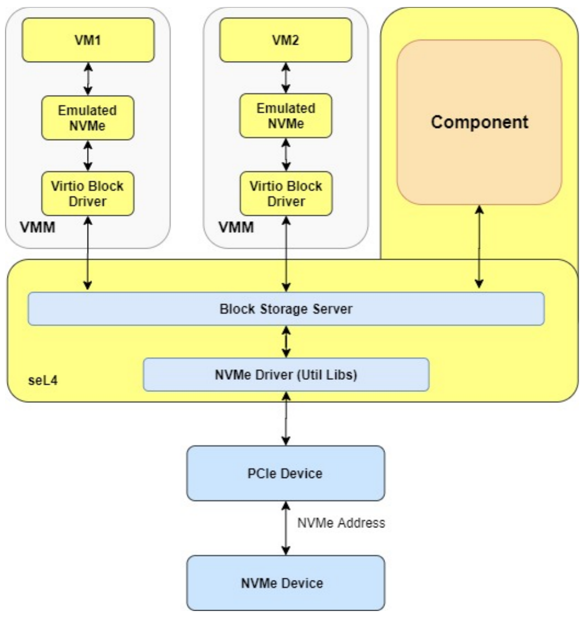


Figure 9: Virtual NVMe Driver

The NVMe virtual driver architecture consists of passing through the address of the NVMe device to PCIe. The PCIe interfaces with an NVMe driver. The block storage server handles API calls from VMMs and components to handle specific commands (i.e. Read, Write, Status, Capacity, etc.). This architecture can be configured to allow for each component and guest VM to have full access to an NVMe drive and/or configured to access only a partition of the NVMe device.

3.3.3 CAN

The CAN seL4 driver consists of a backend CAN Server which is responsible for interfacing with the CAN interface for sending and receiving data on the bus. The CAN server interfaces with the high-level Virtual CAN device which is comprised of two queues for TX/RX traffic. The data from the TX/RX queue feeds into the TX/RX registers of each guest VM. The virtual CAN driver is illustrated in Figure 10.

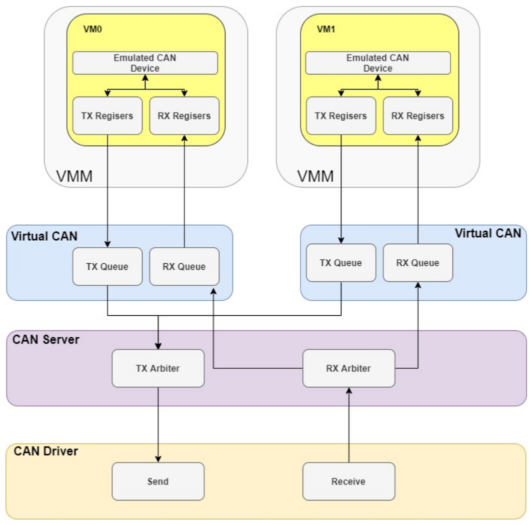


Figure 10: Virtual CAN Driver

3.3.4 Ethernet

Like the implementation for GPIO and NVMe, network device support interfaces a physical Ethernet device to an Ethernet driver. The Ethernet driver enables a connection to VM's via a virtIO-Net interface. Components can interface with the

Ethernet driver directly. Figure 11 illustrates the architecture for the implementation of shared network device support.

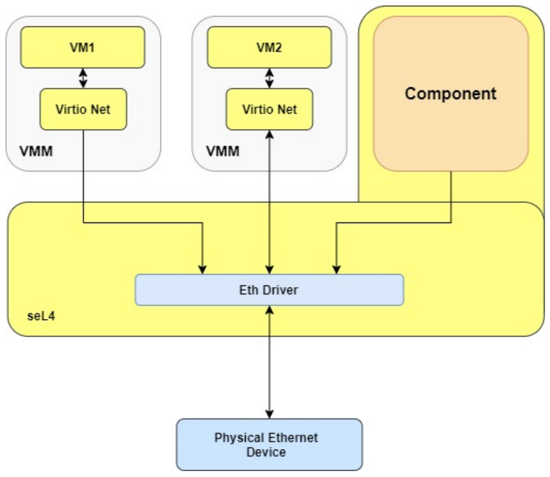


Figure 11: Shared Network Device Support

3.4 TSN Bridging

3.4.1 TSN Standard

TSN bridging between two guest VMs is achieved via a virtIO connection with 802.1Q header support. Packets with an 802.1Q header are switched according to the PCP tag within the header. Untagged packets are switched as best-effort. These are connected to a TSN switch which handles inter-VM communication and communications with the physical NIC, as well as configuration of the TSN IP. The TSN switch can be configured with the allowed Quality of Service (QoS) for each of the connected devices.

3.4.2 Reference Design TSN Implementation

For this current design the inter-VM communications will not conform to TSN standards unless TSN standards are implemented within virtIO-Net. The TSN switch is currently only a switch and does not conform to standards that make it a full TSN bridge. Figure 3 in Section 2.5 illustrates the current implementation of TSN for this effort. For the SmartIO platform there are two VMs implemented; the lower assurance GUI driver and the higher assurance safety critical I/O processor. The high assurance domain has device passthrough access to the TSN device. TSN traffic can be routed from the high assurance domain via a virtIO connection. This type of implementation introduces packet latency between VMs but can be improved by pinning the TSN switch to a single core or by using ePBF or XDP; these two approaches will not completely solve the latency problem without implementing a full TSN bridge.

3.4.3 Advantages

The arbitration implementation required less effort and provided the high assurance VM to have a fully TSN compliant interface. The network topology is easily configurable in software via command line interface (CLI). Complex network topologies are achievable with kernel-based software bridges. Finally, there are no additional components/VMs needed for network filtering/monitoring – this can all be achieved by the arbitration VM.

3.4.4 Disadvantages

Disadvantages to this approach include:

- Requires resource overhead of an entire VM.
- VM kernel adds additional latency.

3.5 Fault Tolerance/Handling

The health monitoring design consisted of 5 elements: VMM Fault Handler, Cyber Monitor Component, Logger Component, Network Application, Host PC. Figure 12 illustrates all 5 stages of the health monitoring process.

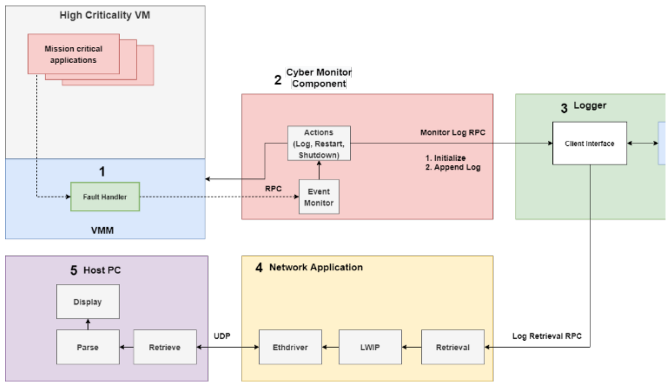


Figure 12: Health Monitoring Architecture

Each time a system event occurs that information is transmitted to the "Cyber Monitor Component" (CMC) executing in user-space via an RPC call. The CMC logs the event and takes necessary action based on the event severity. A separate logging component executing in user-space interfaces with the CMC and stores log information. Logs are retrieved via a network application. The network application is a separate user-space application with a lightweight network stack capable of receiving incoming request via UDP connection from a Host PC.

3.6 VM Secure Boot

The secure boot process implemented in this reference architecture was motivated by the following common threats:

- Local user-initiated installation of malicious system boot firmware
- Malware exploit of weak boot firmware security controls or exploit of system boot firmware through overwrite or modification.
- Network-based system management tools with valid access control (AC)
- Any of the preceding mechanisms could be used to rollback to an authentic but vulnerable system boot firmware

With the preceding attack vectors in mind the security principles that guided this implementation included a process for verifying that the boot firmware image was generated by an authentic source and a mechanism for ensuring that the boot firmware is protected from modification outside of the secure update process. These guidelines were achieved on the Zynq through the Hardware Root of Trust (HWRoT) feature on the hardware by programming the eFUSE. Programming the eFUSE on the platform forces the Configuration Security Unit (CSU) to perform authentication using the asymmetric RSA public/private key scheme. Figure 13 illustrates the signing overview of software images.

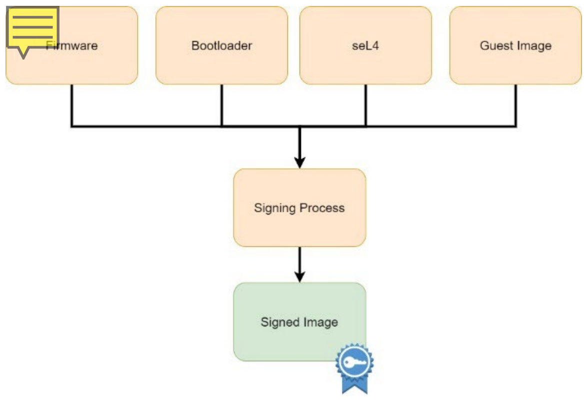


Figure 13: Secure Boot Signing Overview

Subsequent authentication and loading of the seL4 kernel, VMMs, and VMs were achieved by key chaining and deployment of a crypto-component capable of performing RSA authentication. Root of Trust (RoT) persisted by authenticating the previous boot stage before proceeding to the next stage. If at any stage, the authentication process fails, the entire boot process is halted. The verification process is illustrated in Figure 14 below.

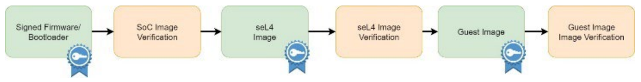


Figure 14: Secure Boot Verification Overview

3.7 Challenges

The specific challenges faced when implementing this architecture were largely platform specific. The process for porting this architecture to different platforms that utilize the Zynq provided hardware dependency hurdles specific to the device tree.

Specifically, when porting to the SmartIO platform there was a challenge in identifying which specific nodes of the device tree were needed for IO card passthrough. Several nodes within the device tree had unintuitive dependencies on other nodes.

4. Testing

Benchmarking results for this effort involved exercising basic functionality of each feature to demonstrate a proof of concept. The intention was to create a full featured platform rather than provide a full featured suite of software components. This approach allowed each software element to be designed with modularity and flexibility in mind so that they can easily adapt and port across platforms/projects within GVSC and the greater DoD.

4.1 TSN Bridging

Initial TSN results tested for throughput of the high priority queues and best effort queues. Table 1 provides the results of benchmarking results.

Table 1: TSN Benchmarking results

Configuration	Bandwidth (Mbits/sec)
High Priority Passthrough Device	871
Low Priority Bridged	886
Best Effort Bridged	227

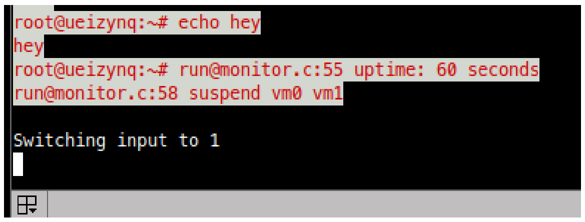
These results indicate that the reference architecture is not achieving optimal performance for a design intending to utilize TSN network traffic. The performance most definitely suffers from a degree of performance degradation because of this architecture. Further investigation would be needed to resolve the low bandwidth performance with future improvements.

4.2 Health Monitoring

Fault handling verification came down to demonstrating four features:

- VM Pause
- VM Reboot
- VM Kill
- Logging/Retrieval

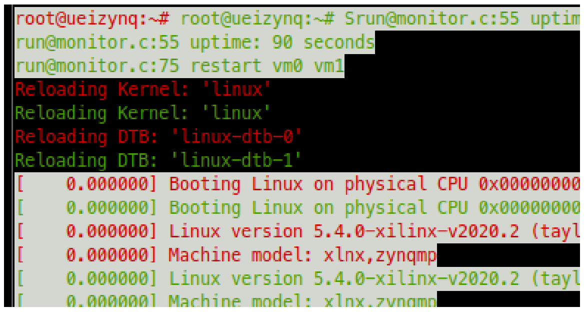
The results of this demonstration were captured via the serial console of VM0 and VM1. For example, Figure 15 below demonstrates the health monitor's capability to pause both VM0 and VM1 by issuing a suspend command.



```
root@ueizynq:~# echo hey
hey
root@ueizynq:~# run@monitor.c:55 uptime: 60 seconds
run@monitor.c:58 suspend vm0 vm1

Switching input to 1
```

Figure 15: VM Pause

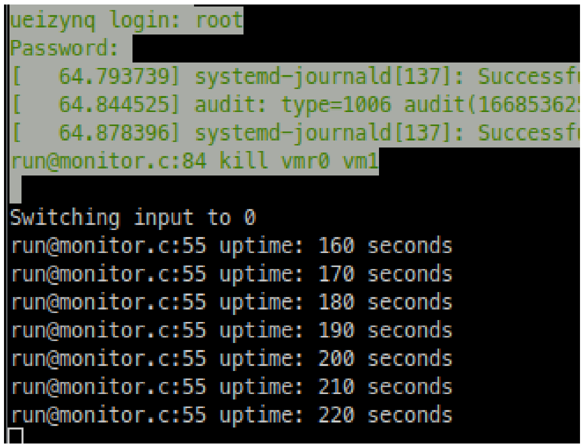
Figure 15 above indicates that after a suspend command has been issued to VM0 and VM1 the serial console ceases printing any information even when sending data to VM0 and VM1 over the serial port. The next demonstration invoked the health monitor's reboot capability.



```
root@ueizynq:~# root@ueizynq:~# Srun@monitor.c:55 uptim
run@monitor.c:55 uptime: 90 seconds
run@monitor.c:75 restart vm0 vm1
Reloading Kernel: 'linux'
Reloading Kernel: 'linux'
Reloading DTB: 'linux-dtb-0'
Reloading DTB: 'linux-dtb-1'
[    0.000000] Booting Linux on physical CPU 0x00000000
[    0.000000] Booting Linux on physical CPU 0x00000000
[    0.000000] Linux version 5.4.0-xilinx-v2020.2 (tayl
[    0.000000] Machine model: xlnx,zynqmp
[    0.000000] Linux version 5.4.0-xilinx-v2020.2 (tayl
[    0.000000] Machine model: xlnx,zynqmp
```

Figure 16: VM Restart

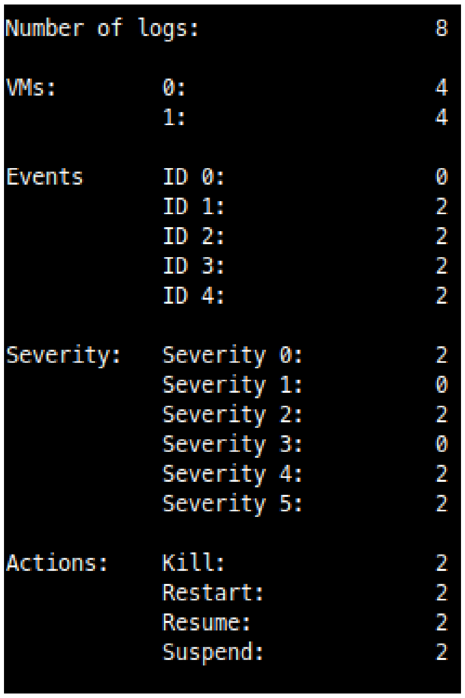
A VM restart command is demonstrated in Figure 16. It can be seen from the serial console output that once restart has been invoked, a new boot trace now appears for VM0 and VM1. The third command supported by the health monitor API is the VM Kill command. This command takes an argument that indicates which VM(s) should cease execution. An example is given in Figure 17.



```
ueizynq login: root
Password:
[   64.793739] systemd-journald[137]: Successf
[   64.844525] audit: type=1006 audit(16685362
[   64.878396] systemd-journald[137]: Successf
run@monitor.c:84 kill vmr0 vm1

Switching input to 0
run@monitor.c:55 uptime: 160 seconds
run@monitor.c:55 uptime: 170 seconds
run@monitor.c:55 uptime: 180 seconds
run@monitor.c:55 uptime: 190 seconds
run@monitor.c:55 uptime: 200 seconds
run@monitor.c:55 uptime: 210 seconds
run@monitor.c:55 uptime: 220 seconds
```

Figure 17: VM Kill

The final feature of the health monitor that was verified was the logging capability. This feature was implemented as a proof of concept to provide real time health metrics of this representative system Figure 18 below provides an example output of system events that occurred during testing.



```
Number of logs:                8

VMs:          0:               4
              1:               4

Events        ID 0:            0
              ID 1:            2
              ID 2:            2
              ID 3:            2
              ID 4:            2

Severity:     Severity 0:      2
              Severity 1:      0
              Severity 2:      2
              Severity 3:      0
              Severity 4:      2
              Severity 5:      2

Actions:      Kill:            2
              Restart:         2
              Resume:          2
              Suspend:         2
```

Figure 18: Health Monitor Log Output

This demonstration provided output for health monitoring of two VMs. Events are appended to the log when they occur and are classified by severity as it relates to survivability of the system. As of right now these classifications are based on arbitrary decisions and are not currently representative of any hard requirement for cyber physical systems being currently developed within DEVCOM-GVSC. However, what the health monitoring results indicate is that this reference architecture now has a portable module that can evolve to suit whatever introspection/health monitoring needs a DEVCOM-GVSC project might have.

4.3 Secure Boot

Secure boot was tested at the following stages: firmware, bootloader, seL4 monolithic image, and guest. The firmware stage was validated by utilizing the CSU of the Zynq architecture. To demonstrate this functionality a hash of the public key used to sign firmware images was flashed to the eFUSE on a development kit. Figure 19 illustrates an overview of how the value stored in eFUSE establishes a RoT for securely booting firmware images.

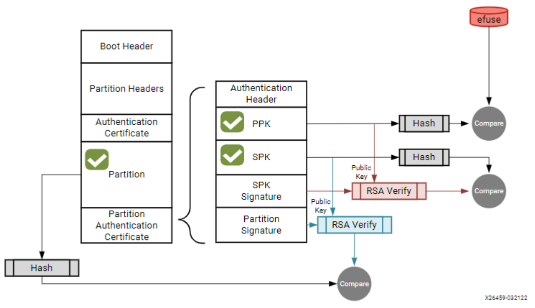


Figure 19: HWRoT with CSU[9]

To test secure boot at the firmware stage a boot image was generated and signed. Using a hexadecimal file editor, a modification was made to the raw data of software partition included in the boot image. This corruption was intended to validate that the CSU can detect changes to the integrity of the boot image and halt the boot process as intended. The only system feedback provided by the development board to indicate that the system failed to boot is a red LED when attempting to boot the corrupted image.

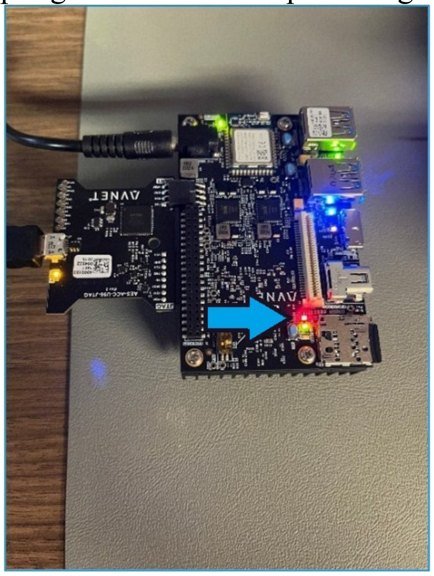


Figure 20: Boot Firmware Corruption Test

Once verification of the CSU was performed the testing then transitioned to validating the bootloader stage. In this case the boot-chain utilized U-Boot to validate signed Flattened-Image-Tree (FIT) images [7]. The FIT image generated for this effort contained the entire monolithic seL4 image. For testing purposes, a similar corruption test was performed to the FIT image to validate that U-Boot would halt the boot process if the integrity of the FIT image couldn't be verified. Finally, the last stage was tested in a similar fashion to the firmware and bootloader. The guest images were programmatically corrupted during configuration/compile time. Table 2 provides the complete set of testing for secure boot in this effort.

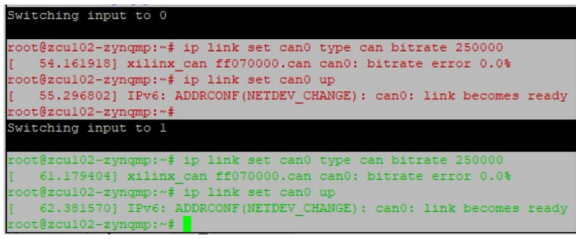
Table 2: Secure Boot Test Cases for Boot Stages

Test	Image	Result
Corruption	Firmware	Boot Halted
	Bootloader	
	Monolithic seL4	
	Guest	
Bad Public Key	Firmware	Boot Halted
	Bootloader	
	Monolithic seL4	
	Guest	
Unsigned Image	Firmware	Boot Halted
	Bootloader	
	Monolithic seL4	
	Guest	
Successful Signed Boot	Firmware	System Booted
	Bootloader	
	Monolithic seL4	
	Guest	

Validation of common use cases: Corruption, Bad Public Key, and Unsigned images were tested at each boot stage. The results were the same for each test case in that the intended result was for the system to halt the boot process and provide meaningful feedback via serial/debug port.

4.4 CAN

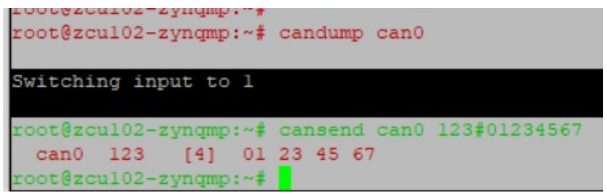
CAN testing consisted of deploying two guest VMs with the virtual CAN driver enabled. This allowed both VMs to be configured to interface with the physical CAN device on a ZCU102. The ZCU102 was selected mainly due to the lack of a CAN interface on the SmartIO platform out of the box. Figure 21 demonstrates both guest VM's being configured to interface with the CAN device.



```
Switching input to 0

root@zcu102-zynqmp:~# ip link set can0 type can bitrate 250000
[   54.161918] xilinx_can ff070000.can can0: bitrate error 0.0%
root@zcu102-zynqmp:~# ip link set can0 up
[   55.296802] IPv6: ADDRCONF(NETDEV_CHANGE): can0: link becomes ready
root@zcu102-zynqmp:~#
Switching input to 1

root@zcu102-zynqmp:~# ip link set can0 type can bitrate 250000
[   61.179404] xilinx_can ff070000.can can0: bitrate error 0.0%
root@zcu102-zynqmp:~# ip link set can0 up
[   62.381570] IPv6: ADDRCONF(NETDEV_CHANGE): can0: link becomes ready
root@zcu102-zynqmp:~#
```

Figure 21: Guest VMs Configured for CAN

From Figure 21 the serial output guest VM0 (in red) readying the uplink for the can0 interface. VM1 (in green) is also demonstrating the ability to enumerate an uplink with the can0 interface. This demonstrates that both VMs are enumerating a CAN device during boot time. Figure 22 below demonstrates a scenario where guest VM0 is configured to receive CAN data transmitted by VM1 using the "cansend" command from Linux.



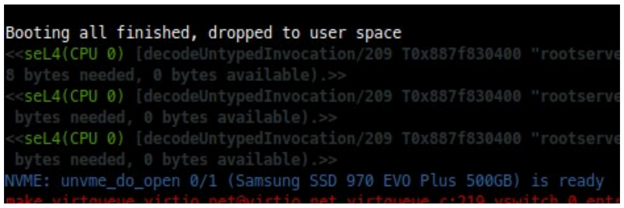
```
root@zcu102-zynqmp:~# candump can0

Switching input to 1

root@zcu102-zynqmp:~# cansend can0 123#01234567
  can0  123   [4]  01 23 45 67
root@zcu102-zynqmp:~#
```

Figure 22: Demonstrating CAN TX/RX from Guest VMs

Figure 22 above demonstrates the ability for guest VM0 (in red) receiving data sent from guest VM1 (in green) and dumping that data to the serial console.

4.5 NVMe

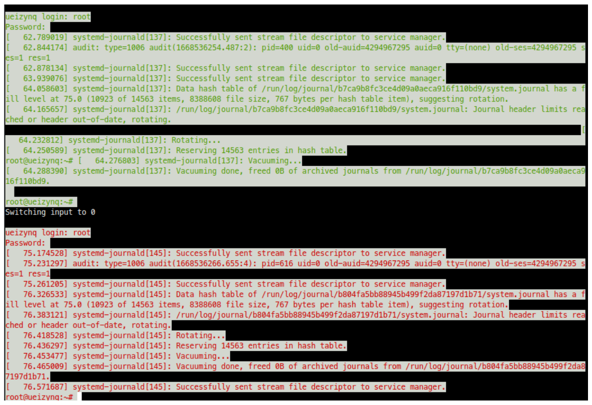
Virtual NVMe driver verification was performed on the SmartIO hardware by installing a 500GB SSD M2 hard drive into the device. Driver support was enabled for seL4 and guest VMs during compile time. Figure 23 below demonstrates a block server interfacing with the SSD during boot time.



```
Booting all finished, dropped to user space
<<seL4(CPU 0) [decodeUntypedInvocation/209 T0x887f830400 "rootserve
8 bytes needed, 0 bytes available).>>
<<seL4(CPU 0) [decodeUntypedInvocation/209 T0x887f830400 "rootserve
 bytes needed, 0 bytes available).>>
<<seL4(CPU 0) [decodeUntypedInvocation/209 T0x887f830400 "rootserve
 bytes needed, 0 bytes available).>>
NVME: unvme_do_open 0/1 (Samsung SSD 970 EVO Plus 500GB) is ready
```

Figure 23: NVMe Verification

During the boot sequence of seL4 the block server component is loaded into memory and begins execution. From Figure 23 above the text output in the serial console indicates that "unvme_do_open" has been invoked and descriptive data of the drive is provided. Furthermore, once the seL4 kernel and respective components completed boot the guest VMs proceeded to boot to separate root file systems. Each VM was given a dedicated partition to boot from and can be observed in Figure 24 below.



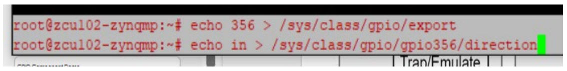
```
ueizynq login: root
Password:
[   62.789019] systemd-journald[137]: Successfully sent stream file descriptor to service manager.
[   62.844174] audit: type=1006 audit(1668536254.487:2): pid=400 uid=0 old-auid=4294967295 auid=0 tty=(none) old-ses=4294967295 s
es=1 res=1
[   62.878134] systemd-journald[137]: Successfully sent stream file descriptor to service manager.
[   63.939076] systemd-journald[137]: Successfully sent stream file descriptor to service manager.
[   64.058603] systemd-journald[137]: Data hash table of /run/log/journal/b7ca9b8fc3ce4d09a0aeca916f110bd9/system.journal has a f
ill level at 75.0 (10923 of 14563 items, 8388608 file size, 767 bytes per hash table item), suggesting rotation.
[   64.165657] systemd-journald[137]: /run/log/journal/b7ca9b8fc3ce4d09a0aeca916f110bd9/system.journal: Journal header limits rea
ched or header out-of-date, rotating.
                                                                                                                                  [
   64.232812] systemd-journald[137]: Rotating...
[   64.250589] systemd-journald[137]: Reserving 14563 entries in hash table.
root@ueizynq:~# [   64.276803] systemd-journald[137]: Vacuuming...
[   64.288390] systemd-journald[137]: Vacuuming done, freed 0B of archived journals from /run/log/journal/b7ca9b8fc3ce4d09a0aeca9
16f110bd9.

root@ueizynq:~#
Switching input to 0

ueizynq login: root
Password:
[   75.174528] systemd-journald[145]: Successfully sent stream file descriptor to service manager.
[   75.231297] audit: type=1006 audit(1668536266.655:4): pid=616 uid=0 old-auid=4294967295 auid=0 tty=(none) old-ses=4294967295 s
es=1 res=1
[   75.261205] systemd-journald[145]: Successfully sent stream file descriptor to service manager.
[   76.326533] systemd-journald[145]: Data hash table of /run/log/journal/b804fa5bb88945b499f2da87197d1b71/system.journal has a f
ill level at 75.0 (10923 of 14563 items, 8388608 file size, 767 bytes per hash table item), suggesting rotation.
[   76.383121] systemd-journald[145]: /run/log/journal/b804fa5bb88945b499f2da87197d1b71/system.journal: Journal header limits rea
ched or header out-of-date, rotating.
[   76.418528] systemd-journald[145]: Rotating...
[   76.436297] systemd-journald[145]: Reserving 14563 entries in hash table.
[   76.453477] systemd-journald[145]: Vacuuming...
[   76.465009] systemd-journald[145]: Vacuuming done, freed 0B of archived journals from /run/log/journal/b804fa5bb88945b499f2da8
7197d1b71.
[   76.571687] systemd-journald[145]: Successfully sent stream file descriptor to service manager.
root@ueizynq:~#
```

Figure 24: Guest VMs Booting from Separate File Systems

This capability allows the reference architecture to further isolate guest VMs by granting them dedicated non-volatile storage.

4.6 GPIO

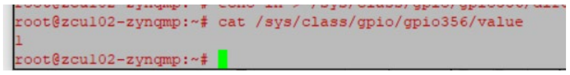
Virtual GPIO driver verification was demonstrated by performing read and write actions that interfaced with physical hardware on the ZCU102 development kit. Guest VM0 was configured to interface with a physical pushbutton on the ZCU102. This is demonstrated in Figure 25 below.



```
root@zcu102-zynqmp:~# echo 356 > /sys/class/gpio/export
root@zcu102-zynqmp:~# echo in > /sys/class/gpio/gpio356/direction
```

Figure 25: Pushbutton Configuration from VM0

Sysfs was used to perform a read of the GPIO pushbutton. When the pushbutton was pressed a corresponding value of 1 appears that indicates a pushbutton press.



```
root@zcu102-zynqmp:~# cat /sys/class/gpio/gpio356/value
1
root@zcu102-zynqmp:~#
```

Figure 26: Push Button Read

Write functionality was demonstrated by toggling an LED on the ZCU102 from guest VM1. Sysfs was used again in this case to write to the corresponding GPIO pin. Figure 27 below demonstrates the process of writing to the corresponding GPIO pin and the LED toggling ON.

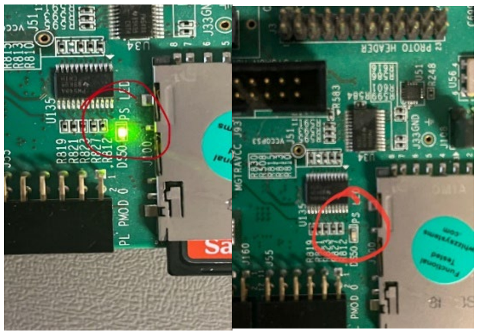


Figure 27: Toggling LED from VM1

What these two demonstrations indicate is that basic read and write functionality from the reference architecture are enabled when using the virtual GPIO driver.

5. Transition Results

Multiple programs at DEVCOM-GVSC including Military Vehicle Hypervisor, (MVH), Cybersecurity for Robotic & Autonomous System Hardening (CRASH) and Enhanced-Vetronics (E-Vetronics)) are currently implementing and validating aspects of the reference architecture covered in this article. Specific functionality for each program is summarized in Table 3 below.

Table 3: Ported Functionality to GVSC Programs

DEVCOM GVSC Program	**High/Low Assurance Domains**	**Shared Driver Support**	**Secure Boot**	**Platform**
Military Hypervisor	X	X		ARMv8
E-Vetronics	X	X	X	ARMv8
CRASH		X		x86

CRASH is utilizing the block server implementation highlighted in this effort. E-Vetronics is utilizing the high/low assurance partitioning, shared driver support, and secure boot. Finally, MVH is also using high/low assurance partitioning along with shared driver support (CAN).

6. Future Improvements

TSN benchmarking indicated that there is a need to investigate how to obtain performance gains within the seL4 microkernel. Naturally, due to virtualization, there is an expectation of some performance degradation with respect to TSN benchmarking; however, the results indicate that beyond the effects of virtualization there are performance degradations because of the seL4 kernel. It is currently up for investigation to determine the root cause and initial discussions seem to indicate that the interrupt framework is the likely candidate to start such work.

As of now this implementation does not provide a way to boot into a failsafe mode if authentication fails during secure boot. One improvement to this design could include the option to boot from eMMC with the option to include a fallback image in the event of authentication failure. In the same vein the addition of a Trusted Platform Module (TPM) would provide the ability for a system architect to deploy measured boot. Measured boot enables the embedded system to have a configurable secure boot mode that allows for the logging of failures to authenticate with the option to halt/proceed boot. A TPM also provides a system designer the means to implement advanced health monitoring techniques: remote attestation and real time introspection to executing processes.

7. REFERENCES

[1] G. Heiser, “Virtualizing embedded systems - why bother?” in 2011 48th ACM/EDAC/IEEE DAC, 2011, pp. 901–905.

[2] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, “Virtualization on TrustZone-Enabled Microcontrollers? Voila!” in ` IEEE RTAS, 2019, pp. 293–304.

[3] T. Prins, “Containerization in Trusted Computing,” In Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium (GVSETS), NDIA, Novi, MI, Aug. 16-18, 2022.

[4] seL4 Foundation, “About seL4”, seL4 Foundation, [Online]. Available: https://sel4.systems/About/.

[5] DornerWorks, Ltd. “Run Your Mixed Criticality Applications Together, Without Interruption, Even When One Crashes,” DornerWorks, Ltd. 9-22-2020. [Online].

[6] R. VanVossen, J. Millwood, C. Guikema, L. Elliott, and J. Roach, "The seL4 Microkernel--A Robust, Resilient, and Open-Source Foundation for Ground Vehicle Electronics Architecture," in *the Ground Vehicle Systems Engineering and Technology Symposium.*

[7] Das U-Boot, “U-boot Documentation” [Online] https://u-boot.readthedocs.io/en/latest/

[8] “Ground Common Infrastructure Architecture” [Online] https://www.dsp.dla.mil/Portals/26/Documents/Conference/2022%20Briefings/Cichosz_DSP%20Conference%20Panel_Distro%20A.pptx?ver=qrtKKCZL4aPXiI527Qay_Q%3D%3D

[9] “Bootgen User Guide v2022.2” Accessed: Jun 6. 2023. [Online]. Available: https://www.xilinx.com/support/documents/s

w_manuals/xilinx2022_1/ug1283-bootgen-user-guide.pdf