2022 NDIA MICHIGAN CHAPTER
GROUND VEHICLE SYSTEMS ENGINEERING
AND TECHNOLOGY SYMPOSIUM
CYBERSECURITY OF GROUND SYSTEMS TECHNICAL SESSION
AUGUST 16-18, 2022 - NOVI, MICHIGAN

Containerization in Embedded Trusted Computing

Taylor Prins[1], Robert VanVossen[1], Tom Barnett[2], Leonard Elliott[3]

[1]DornerWorks, Grand Rapids, MI
[2]CCDC-AvMC, Huntsville, AL
[3]CCDC-GVSC, Warren, MI

ABSTRACT

Interest in application containerization has been on the rise in recent years within the embedded and secure computing communities. Containerization within embedded systems is still relatively new and thus the question of its practical use in secure environments is still unanswered. By using proven kernels and virtual machines, containerization can help play a key role in application development and ease of deployment within trusted computing environments.

Containerization can bring many benefits to the development and deployment of secure applications. These benefits range between ease of development and deployment through use of unified environments to security benefits of namespaces and network isolation. When combined with the seL4 microkernel and DornerWorks use of the VM Composer toolset, mixed criticality systems incorporating containerization can be rapidly and easily developed and deployed to embedded hardware. This paper describes the various advantages, use-cases, and challenges associated with containerization and its use on the mathematically proven seL4 microkernel.

Citation: 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.

1. INTRODUCTION

Vehicle cybersecurity is essential to national security for both military mission assurance in multi domain operations and the commercial movement of people, goods and services in contested environments (in much the same way as the Merchant Marines on the sea). While Army ground vehicles were not designed with cyber threats in mind, they were built to be reliable, safe, and survivable so there is a degree of intrinsic resilience to cyber threats and this buys us the time to address the fundamental challenges to vehicle security. The cyber threat landscape is constantly evolving and we know that unknown vehicle vulnerabilities exist on today's platforms. With increasing software complexity, electrification, autonomy and connectivity, preventative and/or purely defensive approaches will prove to be ineffective and non-enduring. While the due diligence of traditional cyber assurances and penetration testing are valuable to today's fielded systems, the embedded world is adopting techniques like intrusion detection and analytics along with more sophisticated encryption and hashes to provide better security. While these mechanisms, borrowed from the enterprise computing world, bring additional "defense in depth" and awareness, there is a need to think outside the cyber security box and design in security at a more foundational level.

Other challenges with more traditional cyber security approaches involve the scope and scale of the military vehicle fleet and supporting infrastructure. Today there are over a half million platforms in the Army's ground vehicle fleet across the globe and there is no ubiquitous "internet connection" across which to push patches and software updates. Once again, these challenges call for more inherently secure operational technologies. To solve the ground vehicle cyber protection and resiliency challenges, we must fundamentally redesign our vehicle compute architectures with provably secure operating systems and isolation of safety critical and mission critical functionality from those that are more auxiliary or that are more susceptible to cyber-attack.

In that spirit, a proven secure kernel, such as the seL4 microkernel, deployed in combination with software containerization technologies, may provide ground vehicle system developers with an optimal balance of security and flexibility. Secure kernels can be used to provide a robust foundation for computing, while containerization technologies can be layered on top to provide additional isolation as well as dependency management features that may help with the logistical challenges of managing software lifecycle in a disconnected tactical environment.

2. CONTAINERIZATION ON SEL4

2.1. Containerization Overview

Containerization is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure [1].

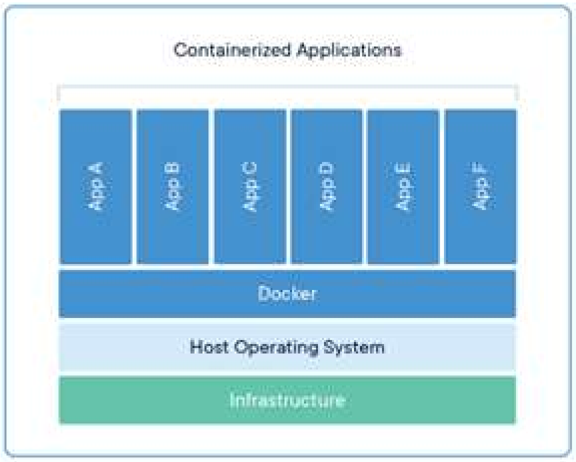

Figure 1: Container Layout [2]

The idea behind a container is that it contains each dependency needed for its application as well as any OS support libraries. This removes the need for the host system to maintain each library and dependency needed for each container's application. This also gives flexibility to the container to use any version of a dependency it needs regardless of what other containers may need. Containers run in isolated instances from each other and the host OS, similar to how virtual machines run on a hypervisor.

Various toolsets exist to manage containers and their operation. Docker (as shown in the above image) is one such toolset that is used synonymously with the word container. In 2013, Docker created an open-source engine that quickly became an industry standard for container management and packaging approach [1]. Today, there are many more options than just Docker and each have their own advantages and disadvantages. Some are targeted more for web and cloud usages allowing mass management and fault-tolerance options whereas others are optimized for resource-constrained embedded systems.

Containers are built from images which are comprised of read-only layers. Each layer contains a certain instruction or set of instructions used to create the container image. When a container is deployed, the top layer is read/write. This separation of layers allows them to be shared across images. For example, say there are two almost identical container images on a system with 5 and 6 layers each respectively. The only difference in the images is that second image has an extra layer that runs a specific command. Physically, there will only be 6 layers stored on the system rather than 11. Because the second container image uses all the same 5 layers as the first, they are reused in the second image instead of duplicated.

2.2. Containerization Advantages

Beyond the use of shared layers, containers have many other advantages. At their core, they are meant to be completely portable and generally platform independent. This portability allows users to develop on one machine and deploy to other hardware platforms with ease. Container dependency isolation gives developers greater freedom with their applications. Being that each container can have its own set of dependencies and OS libraries, updates to applications can be simpler with much less worry on system destabilization and cross compatibility.

There are also many security benefits. At its core, containerization has adopted a "secure-by-default" approach, meaning that security should be inherent in the platform and not a separately deployed and configured solution [1]. This means that the container management engine supports all the same isolation as the underlying OS. This also allows security permissions to be managed by the engine to allow or disallow communications between containers. Namespaces can be used to manage and limit networking, mount points, process IDs, user IDs, inter-process communication, hostname settings, and access to any resources through processes within a container. A subset of this is the networking used by many container management toolsets. Networking can be setup to be completely isolated, shared between containers, shared with the host, or a combination of all three.

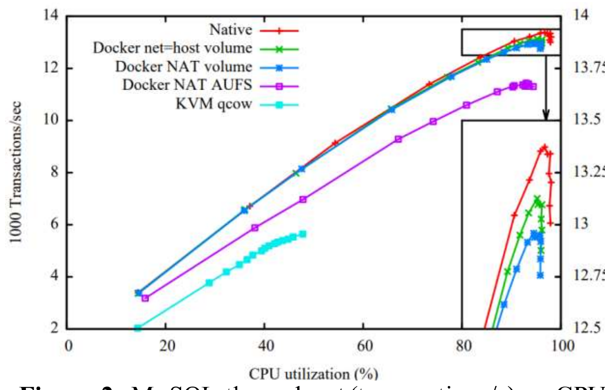

Figure 2: MySQL throughput (transactions/s) vs. CPU utilization. [3].

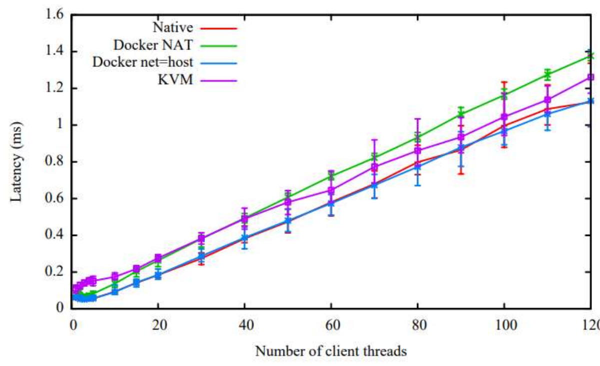

Figure 3: Average latency (in ms) of operations on different Redis deployments. Each data point is the arithmetic mean obtained from 10 runs [3].

Another simple advantage is performance. Although the use of containers comes at the cost of some overhead, application performance has been shown to not be a factor especially when compared to the use of virtual machines for the same applications. Research shows, as seen in Figure 2, that containers introduce negligible overhead for CPU and memory performance (except in extreme cases) [3].

2.3. Containerization Disadvantages

New technology does not come without disadvantages and containers are no different. While containers may have a reduced overhead compared to a virtual machine, there is still RAM overhead and CPU stack space needed for them to run. This may be too much overhead for a simple application with few dependencies. Another downfall is certain networking implementations. As seen in Figure 3, Docker's NAT is slower than running natively or even with a KVM.

Persistent data is less secure and more complex to manage in a containerized system. As discussed earlier, the topmost layer of a container is read/write. When the container shuts down, that layer is removed. Any data that is needed to be persistent across container shutdowns must be held in Docker volumes and mounted inside the container each time. This adds the complexity of managing and securing said volumes.

Virtual machines have been able to run on type 1 hypervisors for decades. Containers need a base OS and management toolset to run and therefore are not optimized to run in a bare metal environment. However, work is being done to mitigate this with a more monolithic and/or unikernal approach for containerized system. This will be discussed later in the paper.

Finally, containerization is currently a fractured ecosystem. Not all containers can be run across the various toolsets. Large and small organizations alike are creating toolsets customized for their needs. While the basis is all the same, the execution and setup can vary [4]. The Open Container Initiative (OCI) was established in 2015 to help unify this fractured ecosystem. It created an open industry standard around container formats and runtimes [10]. Ideally, future universal adaptation would break the fractured ecosystem and make containers and their images truly platform independent.

2.4. seL4 and Containerization

seL4 is a high-assurance microkernel that was built with performance in mind. It is unique because of its comprehensive formal verification, without compromising performance. It is meant to be used as a trustworthy foundation for building safety- and security-critical systems [5]. seL4 has a formal proof of correctness meaning it has a strong isolation story. The microkernel also has a virtual machine monitor (VMM) mode which works in tandem with a VMM user application to provide virtualization. DornerWorks enabled virtualization for seL4 on ARMv8 platforms allowing DEVCOM-GVSC VEA to deploy seL4 in representative military ground vehicle environments. VMM support allowed feature rich software stacks to be run in isolation guaranteed by the seL4 formal proofs [7]. DornerWorks has also proved its versatility and has been able to demonstrate mixed criticality environments within seL4 with secure isolation benefits [6].

Currently, within mixed criticality systems, containerization by itself does not provide strong enough isolation between sub-systems and/or applications. seL4 can be used to separate criticality levels into separate VMs, which can each run various containers, to achieve the desired level of isolation.

This results in a system that has a formally proven/verified microkernel running multiple virtual machines in a mixed-criticality environment in which each VM can run and manage containerized applications and systems. All the benefits of containerization remain with the added security benefits of seL4.

3. Workflow Optimization Through Containerization

Workflows in embedded systems can be a challenge. Long build times with custom environments can be taxing on development cycles. Dependency and library management for larger systems can also be very challenging, this is especially true in a military system where the system lifecycle can span multiple decades. Containerization can help mitigate many of these issues.

Leveraging the platform independent nature of containers significantly improves development productivity. When combined with a tool like DornerWork's VM Composer, developers can quickly spin up seL4 systems utilizing containerization on targeted hardware.

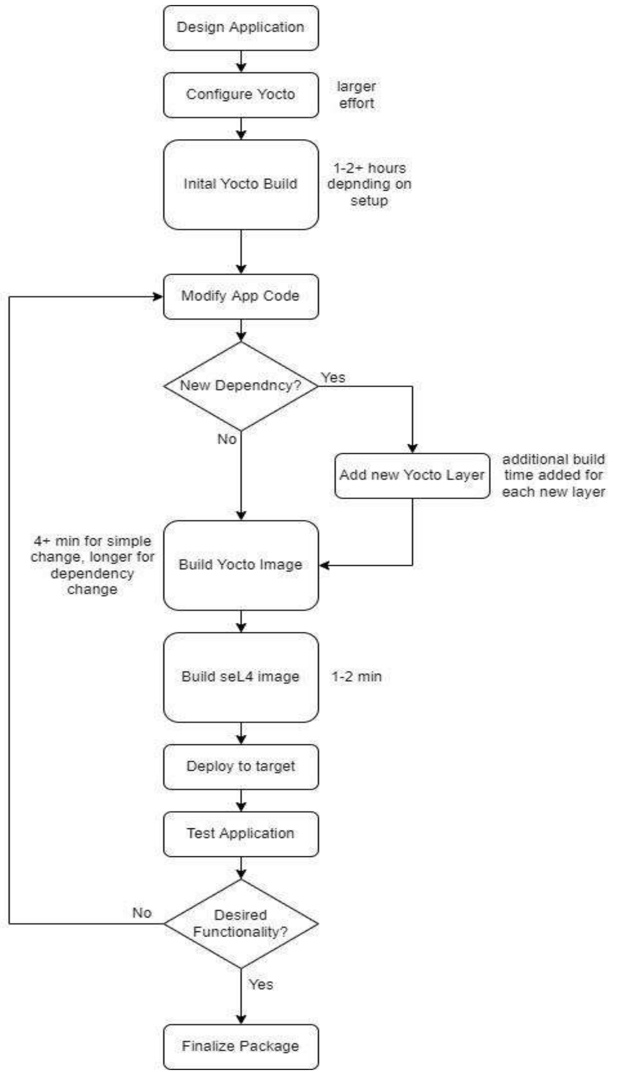

Figure 4: Application development workflow for when using Yocto and deploying with seL4 to target hardware with estimated build times.

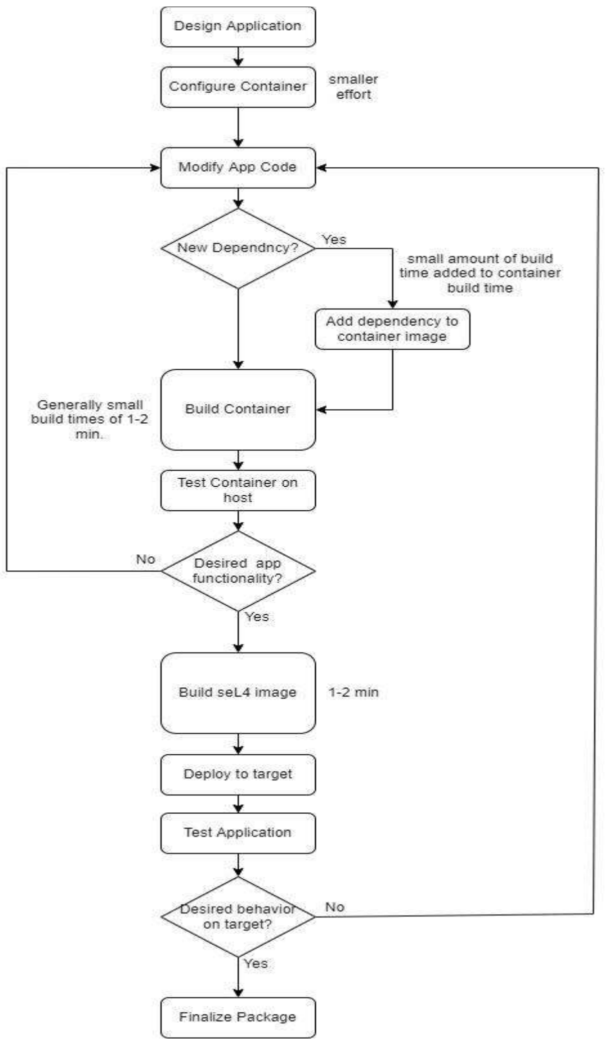

Figure 5: Application development workflow when using containers and VM Composer to build with seL4 and deploy to target hardware with estimated build times.

Figure 4 shows that development cycle times can add up quickly when dealing with complex dependencies for applications. For each new dependency or OS library added in a Yocto build environment, build and management times increase. If very drastic or complex changes are required, it may be required that initial build be rerun thus creating another 1-2+ hour build time. It is also seen that deployment to target hardware and testing must be done on each iteration.

Figure 5 shows that these long build/integration cycle times can be shortened by using containers for application development and VM Composer to package the seL4 image. Due to the platform independence of containers, software can be tested on host/development machines each iteration as well as drastically reducing overall build times. Since each container manages its own dependencies and libraries, the need to manage the overall OS build is reduced. VM Composer packages a prebuilt OS with an OCI container management software (currently Docker) further reducing development efforts for the base OS of the virtual machine.

The use of VM Composer with containers vastly reduces development cycle times and allow the developer to focus on other areas, such as the application development instead of yocto/buildroot image management. The testing process is improved by allowing easier testing of separate applications and allowing tests to be run off the actual target. The outcome of these efforts can produce faster product development cycles and more frequent updates.

4. Army Use-Cases

As mentioned in the introduction, there is a need to isolate safety critical and mission critical functionality from those that are less critical and those more exposed to cyber threats. While physical separation may be an option is some cases, utilization of multi-core processors as well as SWaP constraints drive the need for separation within the compute architecture. In addition, because of the many different vendors, hardware platforms and operating systems, and unique applications and missions across the fleet, the portability of containers and the efficiencies gained throughout the DevSecOps lifecycle may be industry changing.

With these advantages in mind, initial use cases for 2.4. seL4 and containerization would likely be for non-safety critical applications to allow the safety community to assess the approach followed by mixed-criticality (mission, not safety) environments. Finally, the benefits gained through secure containers are not limited to ground vehicles but are also highly extensible other Army platforms in the Aviation and Air & Missile Defense communities.

5. Challenges Towards Utilization

While this approach improves DevSecOps for trusted embedded systems there are still several challenges to overcome. There are only a few container management toolsets that are targeted for embedded use and lower overhead. The most used toolsets are Docker and Kubernetes, which generally have larger resource needs than what would be available for embedded environments. Container images are also much larger due to the tools and libraries they package. Image layers are not always written with resource constraints in mind and thus can produce bloated images. This may only be an issue if using third party and/or ‘off the shelf’ images.

As mentioned previously, the performance of the container applications is near native, but there are still some performance metrics to consider. Boot time of a VM will be negatively impacted by utilizing containers. The container engine needs to run before any of the containers can run. Since the container images can be quite large, the boot time will also be affected by the larger image load times.

A challenge specific to Docker is that of ramdisk usage. Currently Docker has its own set of challenges with running in ramdisk and requires extra configuration to run correctly. This is compounded when attempting to load large images since many embedded systems and platforms are not configured with much more than 4GB of RAM. Systems that need to run in ramdisk, like the seL4 implementations discussed, may run into these same issues. To overcome the ramdisk issues, the obvious approach is to utilize external storage devices for VM filesystems. However, seL4 currently does not have a virtio-blk solution for the virtual machines [8]. This means that each persistent storage device can only be handed to a single VM. This may not be an issue if a platform has a few persistent storage devices and not many VMs with docker support are needed, but a more general approach would be useful.

While these challenges exist today, they should not completely preclude the use of containers on embedded, trusted systems. Some of these can even be mitigated today by using custom container images and lightweight containers. It is also worth noting that not every VM in a system needs to have container support. There is a range of options for each computational piece in the system that makes a trade-off between convenience vs performance/resource utilization. These options are shown in Table 1.

Approach	Overhead/ Resource Utilization	Convenience
seL4 native app (component)	None	Very few pre-existing tools, stacks, and drivers
RTOS VM	Small	More tools, stacks, and drivers
Linux VM	Small-Medium	Lots of tools, stacks, and drivers
Linux VM + Containers	Medium-Large	Same as above + DevSecOps improvements

Table 1: Convenience vs System Impact for different application/VM methods

6. FUTURE ADVANCEMENTS

To make containerization the sought-after solution for embedded systems development, a simplified approach to OS and container management must be accomplished. The current approach can be configured to work for some applications but for others it could add too much overhead/require too many resources. The need for a VM in seL4 adds much more overhead than would be desired to run simple containers. A unikernel and/or simplified virtual machine would be a much simpler design and fit better into the seL4 architecture.

Work is currently being done to simplify container driven VMs as well as port containers to unikernels. Linuxkit is one approach and aims to take containerized systems (built much like a docker-compose file) and create a packaged virtual machine that does nothing but run the containers [9]. This would reduce overhead in many instances but would lose management abilities. Other projects aim to take a container and port it to a unikernel for use with a hypervisor. A container application within a unikernel offers a true monolithic approach but loses some key features of container management that may be useful to the end user and/or developer.

Wind River has made progress introducing containerization into the RTOS environment. The latest versions of their VxWorks RTOS support OCI containers and has container management capabilities [11]. This offers an RTOS with container support and a simplified yet feature rich OS packaged within a small footprint.

Each approach has its own advantages, challenges, and downfalls but all approaches are aimed at the same thing: a simplified container solution for embedded systems while still retaining most of the advantages that come with containerization.

7. REFERENCES

[1] IBM, “Containerization,” IBM, [Online]. Available:

https://www.ibm.com/cloud/learn/containerization.

[2] Docker, “What is a Container?,” Docker, [Online]. Available: https://www.docker.com/resources/what-container.

[3] W Felter, A. Ferreira, R. Rajamony, J. Rubio, “An Updated Performance Comparison of Virtual Machines and Linux Containers,” IBM Research Report, [Online]. Available: https://dominoweb.draco.res.ibm.com/reports/rc25482.pdf. [Accessed 2022]

[4] Channel Futures, “Docker Downsides: Container Cons to Consider before Adopting Docker,” Channel Futures, [Online]. Available: https://www.channelfutures.com/open-source/docker-downsides-container-cons-to-consider-before-adopting-docker.

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

[6] DornerWorks, Ltd. “Run Your Mixed Criticality Applications Together, Without Interruption, Even When One Crashes,” DornerWorks, Ltd. 9-22-2020. [Online]. Available: https://dornerworks.com/blog/sel4-hypervisor-software-isolation-demo/.

[7] 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*.

[8] seL4 Foundation, “libsel4vmmplatsupport”, seL4 Foundation, [Online]. Available: https://docs.sel4.systems/projects/virtualization/libsel4vmmplatsupport.html.

[9] LinuxKit, “LinuxKit”, LinuxKit, [Online]. Available: https://github.com/linuxkit/linuxkit.

[10] Open Container Initiative, “About the Open Container Initiative”, The Linux Foundation, 2020. [Online]. Available: https://opencontainers.org/about/overview/.

[11] M. Chabroux, “RTOS Containers for the Intelligent Edge”, Wind River, 4-26-2021. [Online]. Available: https://blogs.windriver.com/wind_river_blog/2021/04/rtos-containers-for-the-intelligent-edge/.